Universität des Saarlandes
FR 6.2 Informatik
Prof. Dr. Hans-Peter Lenhof
Dipl. Inform. Andreas Hildebrandt

Programmierung II, SS 2003
Musterlösung zur 8. Übung

Aufgabe 1: Laufzeitanalyse (10 Punkte)
Das Aufstellen der Rekurrenz ist in diesem Fall sehr einfach: wir wollen ein Array der Länge n sortieren. Dies kostet uns $T(n)$ Operationen. Zum Sortieren zerlegen wir das Array in ein Teilarray der Länge $n-1$, welches wir rekursiv sortieren (Kosten: $T(n-1)$) und sortieren dann das noch verbleibende Element v_0 in das sortierte Teilarray ein. Dazu laufen wir von vorne über das Array bis wir auf das erste Element stossen, welches größer oder gleich v_0 ist, fügen an dieser Stelle v_0 ein und kopieren den Rest des arrays um eine Position nach hinten. Wir müssen also zum Einfügen einmal komplett über das Array marschieren, was Kosten $\Theta(n)$ verursacht.

Damit erhält unsere Rekurrenz die Form:

$$T(n) = T(n-1) + \Theta(n)$$

wobei wir uns nun einen Vertreter aus $\Theta(n)$ aussuchen können, da ja alle Funktionen dieser Klasse gleich schnell wachsen. Wir wählen uns cn und erhalten als Rekurrenz:

$$T(n) = T(n-1) + cn$$

Das Mastertheorem können wir hier offensichtlich nicht anwenden, denn die Rekurrenz ist nicht von der Form $T(n) = aT(\frac{n}{b}) + f(n)$. Daher iterieren wir die Rekurrenz ein paar mal, um zu sehen ob wir die Lösung erkennen können:

$$\begin{aligned}
T(n) &= T(n-1) + cn \\
&= T(n-2) + c(n-1) + cn \\
&= T(n-3) + c(n-2) + c(n-1) + cn \\
&= \dots \\
&= T(1) + \sum_{i=2}^{n} cn \\
&= 1 + \frac{c}{2}(n^2 + n - 1) \\
&\in \Theta(n^2) \blacksquare
\end{aligned}$$

Aufgabe 2: Hashtabellen (10 Punkte)

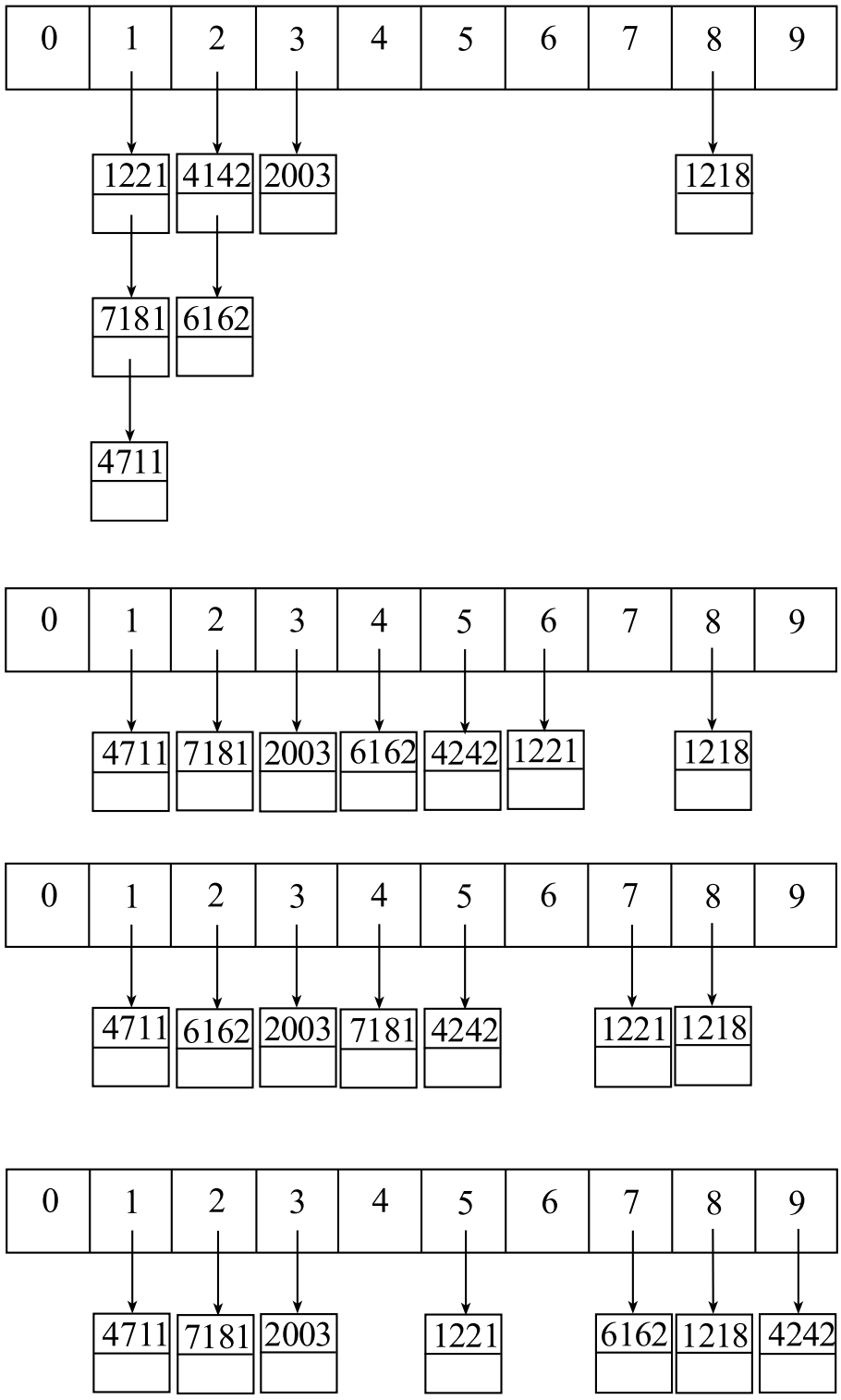

Abbildung 1: Von oben nach unten: Chaining, Linear Probing, Quadratic Probing, Double Hashing

Aufgabe 3: Divisionsmethode (10 Punkte)

- Um zu beweisen, dass eine Menge $\mathcal{H}$ von Funktionen $h : U \to \{0, 1, \ldots, m-1\}$ eine universelle Klasse von Hashfunktionen darstellt müssen wir zeigen, dass es für jedes Paar $x, y \in U, x \neq y$ verschiedener Schlüssel höchstens[1] $\frac{|\mathcal{H}|}{m}$ verschiedene Funktionen $h \in \mathcal{H}$ gibt, die ihnen den gleichen Hashwert $h(x) = h(y)$ zuordnen. Denn in diesem Fall kann man zufällig eine Hashfunktion $h \in \mathcal{H}$ wählen und erhält für die Kollisionswahrscheinlichkeit eines beliebigen Paares ungleicher Schlüssel $x \neq y \in U$ $\frac{1}{m}$. Dies ist genau die Kollisionswahrscheinlichkeit die man erhalten würde, wenn man $h(x)$ und $h(y)$ zufällig aus $\{0, \ldots, m-1\}$ wählen würde, und ein besseres Verhalten können wir von unserer Hashfunktion offensichtlich nicht erwarten.

  Wie können wir nun in unserem Fall die Anzahl von Funktionen aus $\mathcal{H}$ bestimmen, die ein Paar unterschiedlicher Schlüssel auf den gleichen Hashwert abbilden? Da die beiden Schlüssel x, y verschieden sind, müssen sie sich in mindestens einem Bit unterscheiden. Nehmen wir an, die beiden Schlüssel unterscheiden sich $x_0 \neq y_0$ (der Beweis lässt sich ganz analog auf ein beliebiges $x_i \neq y_i$ übertragen. Wir erleichtern uns im Moment nur die Notation). Nach Definition von $h_a(x)$ gilt:

$$
\begin{aligned}
h_a(x) = h_a(y) &\Leftrightarrow h_a(x) - h_a(y) = 0 \\
&\Leftrightarrow \sum_{i=0}^{r} a_i x_i \mod m - \sum_{i=0}^{r} a_i y_i \mod m \equiv 0 \\
&\Leftrightarrow \sum_{i=0}^{r} a_i (x_i - y_i) \mod m \equiv 0 \\
&\Leftrightarrow a_0(x_0 - y_0) \mod m + \sum_{i=1}^{r} a_i (x_i - y_i) \mod m \equiv 0 \\
&\Leftrightarrow a_0(x_0 - y_0) \mod m \equiv -\sum_{i=1}^{r} a_i (x_i - y_i) \mod m
\end{aligned}
$$

  Aus der linearen Algebra ist bekannt, dass eine Menge modulo einer Primzahl einen Körper ergibt, und in einem Körper hat jedes Element ausser dem Nullelement ein eindeutiges Multiplikatives Inverses. Nach der Voraussetzung von oben ist $(x_0 - y_0) \mod m \neq 0$ (denn $x_0 \neq y_0$ und $x_0 < m, y_0 < m$. Damit können wir in der letzten Gleichung oben eindeutig nach $a_0 \mod m$ auflösen. Es gibt also für festgehaltene $a_1, \ldots, a_r$ **genau ein eindeutig bestimmtes** a_0, so dass $h_a(x) = h_a(y)$. Damit folgt, dass für jede zufällige Wahl von $a_1, \ldots, a_r$ genau ein a_0 existiert, welches zu Kollisionen führt. Es gibt genau m^r mögliche Arten, $a_1, \ldots, a_r$ zu Wählen, und damit haben wir gezeigt, dass zwei beliebige Schlüssel $x \neq y \in U$ für genau m^r verschiedene Werte von a und damit für genau m^r verschiedene $h_a \in \mathcal{H}$ kollidieren. Da die Menge $\mathcal{H}$ per Definition genau m^{r+1} Elemente hat (denn genau so viele verschiedene Werte von a können wir bilden), folgt, dass die Wahrscheinlichkeit, dass bei willkürlich gewähltem h_a x und y kollidieren, genau $\frac{m^r}{m^{r+1}} = \frac{1}{m}$ beträgt. Also ist $\mathcal{H}$ eine universelle Klasse von Hashfunktionen.■

[1] Manche Autoren fordern hier anstatt *mindestens genau*

- Was geht nun schief, wenn wir verlangen, dass $a_i \neq 0 \forall i \in 0, \ldots, r$? Naiv könnte man annehmen, dass unsere Klasse von der Einschränkung nur profitieren kann, denn mit $a_i \neq 0 \forall i$ folgt natürlich auch dass $a \neq 0$, und damit ist die schlechtestmögliche Hashfunktion $h_0(x) = 0 \forall x$, die für jedes Schlüsselpaar Kollisionen bewirkt ausgeschlossen. Andere Schlüsselpaare jedoch kollidieren nun deutlich häufiger. Da die Definition von universellem Hashing verlangt, dass die Kollisionswahrscheinlichkeit für alle ungleichen Schlüsselpaare kleiner als $\frac{1}{m}$ ist, genügt es ein einziges Paar zu betrachten, das häufiger zu Kollisionen führt.
  Wählen wir $x :=< 0, \ldots, 0, 1 >$ und $y :=< 0, \ldots, 0, 1, 0 >$, also $x_0 = 1, x_i = 0 \forall i \neq 0$, $y_1 = 1, y_i = 0 \forall i \neq 1$, so gilt offenbar

$$\begin{aligned} h_a(x) = h_a(y) &\Leftrightarrow h_a(x) - h_a(y) = 0 \\ &\Leftrightarrow \sum_{i=0}^{r} a_i(x_i - y_i) \mod m \equiv 0 \\ &\Leftrightarrow a_0 \underbrace{x_0}_{1} - a_1 \underbrace{y_1}_{1} \mod m \equiv 0 \\ &\Leftrightarrow a_0 \equiv a_1 \mod m \end{aligned}$$

  Dies bedeutet, dass in diesem Fall die Wahl von a_1 a_0 schon eindeutig bestimmt. $a_1, \ldots, a_r$ sind natürlich weiterhin völlig frei wählbar, allerdings mit der Einschränkung $a_i \neq 0$. Damit gibt es für die Wahl von a $(m-1)^r$ Möglichkeiten, die zu einer Kollision führen. Auch die Klasse $\mathcal{H}$ ist nun kleiner: sie enthält nämlich nur noch $(m-1)^{r+1}$ Elemente. Damit erhalten wir für die Kollisionswahrscheinlichkeit der oben gewählten Schlüssel x und y:

$$\frac{(m-1)^r}{(m-1)^{r+1}} = \frac{1}{m-1} > \frac{1}{m}$$

  Die Kollisionswahrscheinlichkeit ist nun also zu gross, und $\mathcal{H}$ keine Klasse von universellen Hashfunktionen mehr!

Aufgabe 4: Hashtabellen, Implementierung (10 Punkte)
Den folgenden Code finden Sie in der Datei `HashTabelleChaining.h`:

```
#include <list>
#include <vector>

using namespace std;

// Die HashTabellen-Klasse
template <class Typ, class Hashfunktion>
class HashTabelleChaining
{
  public:
    // Der Konstruktor. Wir brauchen eine
```

```
    // Groesse und eine Hashfunktion (der
    // Templateparameter Hashfunktion ist
    // ja nur ein Datentyp)
    HashTabelleChaining(int m, const Hashfunktion& f);

    // Der Destruktor
    ~HashTabelleChaining();

    // Einfuegen eines Elementes
    void einfuegen(const Typ& element);

    // Entfernen eines Elementes
    bool entfernen(const Typ& element);

    // Suche nach einem Element
    Typ suche(const Typ& element);

  protected:
    // Die Tabellengroesse
    int groesse_;

    // Die Hashfunktion
    Hashfunktion hash_;

    // Die eigentliche Hashtabelle
    vector< list<Typ> > daten_;
};

// Der Konstruktor muss nur die Argumente in die
// Klasse kopieren und die Tablle auf die richtige
// Groesse setzen
template <class Typ, class Hashfunktion>
HashTabelleChaining<Typ, Hashfunktion>::HashTabelleChaining(int m, const Hashfunktion
  : groesse_(m),
    hash_(f),
    daten_(m)
{
}

// Der Destruktor hat nichts zu tun
template <class Typ, class Hashfunktion>
HashTabelleChaining<Typ, Hashfunktion>::~HashTabelleChaining()
{
}

// Einfuegen eines Elementes
template <class Typ, class Hashfunktion>
void HashTabelleChaining<Typ, Hashfunktion>::einfuegen(const Typ& element)
```

```
{
  // Wir pushen das Element vorne in die Liste
  // die zu seinem Hashwert gehoert
  daten_[hash_(element)].push_front(element);
}

// Entfernen eines Elementes. Wir geben noch
// zurueck, ob das entfernen erfolgreich war
// Der Einfachheit halber verwenden wir hier
// wieder einen vollen _Typ_, obwohl ja eigentlich
// der Key ausreichen wuerde
template <class Typ, class Hashfunktion>
bool HashTabelleChaining<Typ, Hashfunktion>::entfernen(const Typ& element)
{
  // Zuerst brauchen wir den Hashwert des Elementes
  int position = hash_(element);

  // Dann suchen wir, ob das Element in der Tabelle
  // enthalten ist
  typename list<Typ>::iterator pos_in_liste;

  pos_in_liste = find(daten_[position].begin(),
      daten_[position].end(), element);

  if (pos_in_liste != daten_[position].end())
  {
    daten_[position].erase(pos_in_liste);
    return true;
  }
  else
  {
    return false;
  }
}

// Und schliesslich noch die Suche in der Tabelle.
// Wir geben das gefundene Element zurueck, da wir
// ja eigentlich nur nach dem Schluessel gesucht haben,
// und den Anwender der Inhalt interessieren wird
template <class Typ, class Hashfunktion>
Typ HashTabelleChaining<Typ, Hashfunktion>::suche(const Typ& element)
{
  // Zuerst wieder der Hashwert
  int position = hash_(element);

  typename list<Typ>::iterator pos_in_list;

  pos_in_list = find(daten_[position].begin(),
```

```
        daten_[position].end(), element);

  if (pos_in_list == daten_[position].end())
  {
    // nicht gefunden... Wir geben ein Defaultkonstruiertes
    // Element vom Typ "Typ" zurueck
    Typ resultat;

    return resultat;
  }
  else
  {
    return *pos_in_list;
  }
}
```

In der Datei `chaining.C` finden Sie ein paar Tests unserer neuen Klasse:

```
#include "HashTabelleChaining.h"

#include <string>
#include <iostream>

using namespace std;


// Ein einfaches Funktionsobjekt
template <class Typ>
class hashmod
{
  public:
    hashmod(int m);
    ~hashmod();

    int operator() (const Typ& element);

  protected:
    int m_;
};

template <class Typ>
hashmod<Typ>::hashmod(int m)
  : m_(m)
{
}

template <class Typ>
hashmod<Typ>::~hashmod()
{
}
```

```
template <class Typ>
int hashmod<Typ>::operator() (const Typ& element)
{
  // Wir verlangen, dass "element" ein Member namens
  // "key" besitzt, welches sich auf int casten laesst
  return (int)element.key % m_;
}

// Ein einfacher Datentyp fuer die Hashtabelle
class Kontakt
{
  public:
    Kontakt();
    Kontakt(string Name, int Nummer);
    ~Kontakt();

    bool operator == (const Kontakt& );
    string name;
    int key;
};

Kontakt::Kontakt()
{
  name = "";
  key  = 0;
}

Kontakt::Kontakt(string Name, int Nummer)
  : name(Name),
    key(Nummer)
{
}

Kontakt::~Kontakt()
{
}

// Diesen Operator brauchen wir fuer "find" in der
// Hashtabelle
bool Kontakt::operator == (const Kontakt& k)
{
  return key == k.key;
}

// Und schliesslich ein paar Tests
int main(int argc, char **argv)
{
```

```
  hashmod<Kontakt> h(10);

  HashTabelleChaining<Kontakt, hashmod<Kontakt> > tabelle(10, h);

  Kontakt k("Test", 1234);
  tabelle.einfuegen(k);
  k.key = 4567;
  tabelle.einfuegen(k);

  const Kontakt k2 = tabelle.suche(k);

  cout << k2.name << " " << k2.key << endl;

  return 0;
}
```

Aufgabe 5. (Hashtabellen, Implementierung II Bonuspunkte)
10 Den folgenden Code finden Sie in der Datei `HashTabelleOA.h`:

```
#include <list>
#include <vector>

using namespace std;

// Die HashTabellen-Klasse
template <class Typ, class Hashfunktion, class Addressfunktion>
class HashTabelleOA
{
  public:
    // Der Konstruktor. Wir brauchen eine
    // Groesse und eine Hashfunktion (der
    // Templateparameter Hashfunktion ist
    // ja nur ein Datentyp)
    HashTabelleOA(int m, const Hashfunktion& f, const Addressfunktion& a);

    // Der Destruktor
    ~HashTabelleOA();

    // Einfuegen eines Elementes
    void einfuegen(const Typ& element);

    // Entfernen eines Elementes
    bool entfernen(const Typ& element);

    // Suche nach einem Element
```

```
    Typ suche(const Typ& element);

  protected:
    // Die Tabellengroesse
    int groesse_;

    // Die Hashfunktion
    Hashfunktion hash_;

    // Die Addressfunktion
    Addressfunktion addr_;

    // Die eigentliche Hashtabelle
    vector<Typ> daten_;

    // Wir machen es uns einfach und markieren die Felder,
    // in denen einmal ein Element geloescht wurde. Das macht
    // die Suche bei open adressing einfacher...
    vector<bool> dirty_;
};

// Der Konstruktor muss nur die Argumente in die
// Klasse kopieren und die Tablle auf die richtige
// Groesse setzen. Dann muss der Vektor mit -1 gef"ullt werden.
// Da wir "-1" nicht verwenden koennen (wir kennen ja "Typ" nicht)
// verwenden wir statt dessen ein default konstruiertes Element
// vom Typ "Typ"
template <class Typ, class Hashfunktion, class Addressfunktion>
HashTabelleOA<Typ, Hashfunktion, Addressfunktion>::HashTabelleOA(int m,
  const Hashfunktion& f, const Addressfunktion& a)
  : groesse_(m),
    hash_(f),
    addr_(a),
    daten_(m),
    dirty_(m)
{
  Typ t;
  for (unsigned int i = 0; i < daten_.size(); i++)
  {
    daten_[i] = t;
    dirty_[i] = false;
  }
}

// Der Destruktor hat nichts zu tun
template <class Typ, class Hashfunktion, class Addressfunktion>
HashTabelleOA<Typ, Hashfunktion, Addressfunktion>::~HashTabelleOA()
{
```

```
}

// Einfuegen eines Elementes
template <class Typ, class Hashfunktion, class Addressfunktion>
void HashTabelleOA<Typ, Hashfunktion, Addressfunktion>::einfuegen(const Typ& element)
{
  // Zun"achst brauchen wir den Hashwert des Elementes
  // die zu seinem Hashwert gehoert
  int position = hash_(element);

  int i = 0;

  Typ t;

  // und jetzt suchen wir ein freies Plaetzchen. Wir verwenden ja
  // Defaultkonstruierte "Typ" Variablen zum Markieren von freien Plaetzen.
  // Wir wollen uns nicht darauf verlassen, dass "Typ" einen operator != implementier
  // und verwenden statt dessen ! (operator == )
  while ( !(daten_[position] == t) )
  {
    position = addr_(element, i);
    i++;
  }

  daten_[position] = element;
  dirty_[position] = false;
}

// Entfernen eines Elementes. Wir geben noch
// zurueck, ob das entfernen erfolgreich war
// Der Einfachheit halber verwenden wir hier
// wieder einen vollen _Typ_, obwohl ja eigentlich
// der Key ausreichen wuerde
template <class Typ, class Hashfunktion, class Addressfunktion>
bool HashTabelleOA<Typ, Hashfunktion, Addressfunktion>::entfernen(const Typ& element)
{
  // Zuerst brauchen wir den Hashwert des Elementes
  int position = hash_(element);

  int i = 0;

  Typ t;

  // und jetzt schauen wir nach, ob es schon enthalten ist
  while (!(daten_[position] == t))
  {
    position = addr_(element, ++i);
  }
```

```
  daten_[position] = t;

  // Wir muessen uns merken dass wir hier ein Element
  // geloescht haben, sonst funktioniert die Suche nicht
  dirty_[position] = true;
}

// Und schliesslich noch die Suche in der Tabelle.
// Wir geben das gefundene Element zurueck, da wir
// ja eigentlich nur nach dem Schluessel gesucht haben,
// und den Anwender der Inhalt interessieren wird
// Suche in einer Hashtabelle mit Open Adressing hat ein
// schwerwiegendes Problem: da wir ja bei Kollisionen
// immer weiter in der Tabelle hin und her springen,
// kann es uns z.B. passieren, dass wir nach 2 Kollisionen
// 3 Elemente mit gleichem Hashkey abgespeichert haben und
// nun das 2. (welches bei open adressing zu i = 1 gehoert)
// entfernen. An dieser Stelle steht dann -1. Bei einer darauffolgenden
// Suche nach dem 3. Element stossen wir zunaechst auf diese
// -1 und koennen nicht entscheiden, ob wir noch weiter
// suchen muessen oder nicht. Dafuer merken wir uns einfach
// bei jedem Feld, ob es einmal geloescht wurde oder nicht.
// Nach einigen Loeschoperationen muessen wir dann natuerlich
// doch wieder die ganze Tabelle durchsuchen, daher wuerde
// man in einer Real-World Implementierung entweder die
// Tabelle regelmaessig reorganisieren (amortisiert) oder
// das Loeschen ein wenig ausgefuchster Implementieren
template <class Typ, class Hashfunktion, class Addressfunktion>
Typ HashTabelleOA<Typ, Hashfunktion, Addressfunktion>::suche(const Typ& element)
{
  // Zuerst brauchen wir den Hashwert des Elementes
  int position = hash_(element);

  int i = 0;

  Typ t;

  // und jetzt schauen wir nach, ob es schon enthalten ist
  while ( !(daten_[position].key == element.key) && !(daten_[position] == t) && (dirt
  {
    position = addr_(element, ++i);
  }


  if (daten_[position].key == element.key)
  {
    return daten_[position];
```

```
  }
  else
  {
    // nicht gefunden... Wir geben ein Defaultkonstruiertes
    // Element vom Typ "Typ" zurueck
    Typ resultat;

    return resultat;
  }
}
```

Und ein paar Tests in der Datei `openaddressing.C`:

```
#include "HashTabelleOA.h"

#include <string>
#include <iostream>

using namespace std;


// Ein einfaches Funktionsobjekt
template <class Typ>
class hashmod
{
  public:
    hashmod(int m);
    ~hashmod();

    int operator() (const Typ& element);

  protected:
    int m_;
};

template <class Typ>
hashmod<Typ>::hashmod(int m)
  : m_(m)
{
}

template <class Typ>
hashmod<Typ>::~hashmod()
{
}

template <class Typ>
int hashmod<Typ>::operator() (const Typ& element)
{
  // Wir verlangen, dass "element" ein Member namens
```

```
  // "key" besitzt, welches sich auf int casten laesst
  return (int)element.key % m_;
}

// Ein Funktionsobjekt fuer linear probing
template <class Typ, class Hashfunktion>
class linearProbing
{
  public:
    linearProbing(int m, const Hashfunktion& h);
    ~linearProbing();

    int operator () (const Typ& k, int i);

  protected:
    int m_;
    Hashfunktion h_;
};

template <class Typ, class Hashfunktion>
linearProbing<Typ, Hashfunktion>::linearProbing(int m, const Hashfunktion& h)
  : m_(m),
    h_(h)
{
}

template <class Typ, class Hashfunktion>
linearProbing<Typ, Hashfunktion>::~linearProbing()
{
}

template <class Typ, class Hashfunktion>
int linearProbing<Typ, Hashfunktion>::operator() (const Typ& k, int i)
{
  return (h_(k) + i) % m_;
}

// Ein Funktionsobjekt fuer quadratic probing
template <class Typ, class Hashfunktion>
class quadraticProbing
{
  public:
    quadraticProbing(int m, const Hashfunktion& h, int c1, int c2);
    ~quadraticProbing();

    int operator () (const Typ& k, int i);

  protected:
```

```
    int m_;
    Hashfunktion h_;
    int c1_, c2_;
};

template <class Typ, class Hashfunktion>
quadraticProbing<Typ, Hashfunktion>::quadraticProbing(int m, const Hashfunktion& h, i
  : m_(m),
    h_(h),
    c1_(c1),
    c2_(c2)
{
}

template <class Typ, class Hashfunktion>
quadraticProbing<Typ, Hashfunktion>::~quadraticProbing()
{
}

template <class Typ, class Hashfunktion>
int quadraticProbing<Typ, Hashfunktion>::operator() (const Typ& k, int i)
{
  return (h_(k) + c1_*i + c_2*i*i) % m_;
}

// Ein Funktionsobjekt fuer quadratic probing
template <class Typ, class Hashfunktion>
class doubleHashing
{
  public:
    doubleHashing(int m, const Hashfunktion& h, const Hashfunktion& h2);
    ~doubleHashing();

    int operator () (const Typ& k, int i);

  protected:
    int m_;
    Hashfunktion h_, h2_;
};

template <class Typ, class Hashfunktion>
doubleHashing<Typ, Hashfunktion>::doubleHashing(int m, const Hashfunktion& h, const H
  : m_(m),
    h_(h),
    h2_(h2)
{
}
```

```
template <class Typ, class Hashfunktion>
doubleHashing<Typ, Hashfunktion>::~doubleHashing()
{
}

template <class Typ, class Hashfunktion>
int doubleHashing<Typ, Hashfunktion>::operator() (const Typ& k, int i)
{
  return (h_(k) + i*h2_(k)) % m_;
}

// Ein einfacher Datentyp fuer die Hashtabelle
class Kontakt
{
  public:
    Kontakt();
    Kontakt(string Name, int Nummer);
    ~Kontakt();

    bool operator == (const Kontakt& );
    string name;
    int key;
};

Kontakt::Kontakt()
{
  name = "";
  key  = 0;
}

Kontakt::Kontakt(string Name, int Nummer)
  : name(Name),
    key(Nummer)
{
}

Kontakt::~Kontakt()
{
}

// Diesen Operator brauchen wir fuer "find" in der
// Hashtabelle
bool Kontakt::operator == (const Kontakt& k)
{
  return key == k.key;
}

// Und schliesslich ein paar Tests
```

```
int main(int argc, char **argv)
{
  hashmod<Kontakt> h(10);
  linearProbing<Kontakt, hashmod<Kontakt> > a(10, h);

  HashTabelleOA<Kontakt, hashmod<Kontakt>, linearProbing<Kontakt, hashmod<Kontakt> >

  Kontakt k("Test", 1234);
  tabelle.einfuegen(k);
  k.key = 4567;
  tabelle.einfuegen(k);

  const Kontakt k2 = tabelle.suche(k);

  cout << k2.name << " " << k2.key << endl;

  return 0;
}
```
