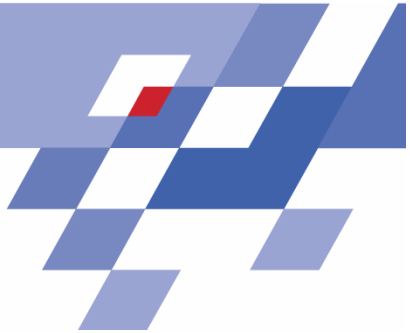
TECHNISCHE UNIVERSITÄT DORTMUND



Wintersemester 2007/08

Einführung in die Informatik für Naturwissenschaftler und Ingenieure

(alias **Einführung in die Programmierung)**

(Vorlesung)

Prof. Dr. Günter Rudolph

Fakultät für Informatik

Lehrstuhl für Algorithm Engineering

Nachtrag zu Kapitel 12: Virtuelle Destruktoren

Wird ein Objekt einer abgeleiteten Klasse über einen Verweis / Zeiger auf die Basisklasse frei gegeben, dann muss der **Destruktor in der Basisklasse virtuell** sein!

Warum?

Wenn nicht virtuell, dann Bindung des Destruktors statisch zur Übersetzungszeit!

$\Rightarrow$ Immer Aufruf des Destruktors der Basisklasse!

Nachtrag zu Kapitel 12: Virtuelle Destruktoren

```
class Familie {
public:
  ~Familie() { cout << “D: Familie“ << endl; }
};

class Sohn : public Familie {
  ~Sohn() { cout << “D: Sohn“ << endl; }
};

class Tochter : public Familie {
  ~Tochter() { cout << “D: Tochter“ << endl; }
};

int main() {
  Familie *fam[3] = { new Familie, new Sohn, new Tochter };
  delete fam[0]; delete fam[1]; delete fam[2];
  return 0;
}
```

Ausgabe:

```
D: Familie
D: Familie
D: Familie
```

Nachtrag zu Kapitel 12: Virtuelle Destruktoren

```
class Familie {
public:
  virtual ~Familie() { cout << “D: Familie“ << endl; }
};

class Sohn : public Familie {
  ~Sohn() { cout << “D: Sohn“ << endl; }
};

class Tochter : public Familie {
  ~Tochter() { cout << “D: Tochter“ << endl; }
};

int main() {
  Familie *fam[3] = { new Familie, new Sohn, new Tochter };
  delete fam[0]; delete fam[1]; delete fam[2];
  return 0;
}
```

Ausgabe:

```
D: Familie
D: Sohn
D: Familie
D: Tochter
D: Familie
```

Inhalt

- Motivation
- Grobentwurf
- ADT List
- ADT HashTable
- Anwendung
- Umstrukturierung des Codes (*refactoring*)

Motivation

Gesucht: Datenstruktur zum Einfügen, Löschen und Auffinden von Elementen

$\Rightarrow$ Binäre Suchbäume!

Problem: Binäre Suchbäume erfordern eine totale Ordnung auf den Elementen

Totale Ordnung

Jedes Element kann mit jedem anderen verglichen werden:

Entweder a < b oder a > b oder a = b. Beispiele: $\mathbb{N}$, $\mathbb{R}$, { A, B, …, Z }, …

Partielle Ordnung

Es existieren unvergleichbare Elemente: a || b

Beispiele: $\mathbb{N}^2$, $\mathbb{R}^3$, …

$$\begin{pmatrix} 2 \\ 5 \end{pmatrix} < \begin{pmatrix} 8 \\ 6 \end{pmatrix}; \quad \begin{pmatrix} 2 \\ 5 \end{pmatrix} \| \begin{pmatrix} 3 \\ 4 \end{pmatrix}$$

Motivation

Gesucht: Datenstruktur zum Einfügen, Löschen und Auffinden von Elementen

Problem: Totale Ordnung nicht auf natürliche Art vorhanden

Beispiel: Vergleich von Bilddaten, Musikdaten, komplexen Datensätzen

$\Rightarrow$ Lineare Liste!

Funktioniert, jedoch mit ungünstiger Laufzeit:

1. Feststellen, dass Element nicht vorhanden: N Vergleiche auf Gleichheit
2. Vorhandenes Element auffinden: im Mittel (N+1) / 2 Vergleiche

(bei diskreter Gleichverteilung)

$\Rightarrow$ Alternative Suchverfahren notwendig! $\Rightarrow$ ***Hashing***

Idee

1. Jedes Element e bekommt einen **numerischen** „Stempel“ h(e), der sich aus dem **Dateninhalt** von e berechnet
2. Aufteilen der Menge von N Elementen in M disjunkte Teilmengen, wobei M die Anzahl der möglichen Stempel ist
   → Elemente mit gleichem Stempel kommen in dieselbe Teilmenge
3. Suchen nach Element e nur noch in Teilmenge für Stempel h(e)

Laufzeit (Annahme: alle M Teilmengen ungefähr gleich groß)

a) Feststellen, dass Element nicht vorhanden: N / M Vergleiche auf Gleichheit

b) Vorhandenes Element auffinden: im Mittel (N / M +1) / 2 Vergleiche

(bei diskreter Gleichverteilung)

⇒ deutliche Beschleunigung!

Grobentwurf

1. Jedes Element e $\in$ E bekommt einen **numerischen** „Stempel“ h(e), der sich aus dem **Dateninhalt** von e berechnet

   Funktion h: E $\rightarrow$ { 0, 1, …, M – 1 } heißt **Hash-Funktion** (*to hash*: zerhacken)

   Anforderung: sie soll zwischen 0 und M – 1 gleichmäßig verteilen

2. Elemente mit gleichem Stempel kommen in dieselbe Teilmenge

   M Teilmengen werden durch M lineare Listen realisiert (ADT List)

   Tabelle der Größe M enthält für jeden Hash-Wert einen Zeiger auf seine Liste

3. Suchen nach Element e nur noch in Teilmenge für Stempel h(e)

   Suche nach e $\rightarrow$ Berechne h(e), hole Zeiger auf Liste aus Tabelle[h(e)],

   Suche in dieser Liste nach Element e

Grobentwurf

Weitere Operationen auf der Basis von „Suchen“

- **Einfügen** von Element e

  → Suche nach e in Liste für Hash-Werte h(e)
  Nur wenn e **nicht** in dieser Liste, dann am Anfang der Liste einfügen

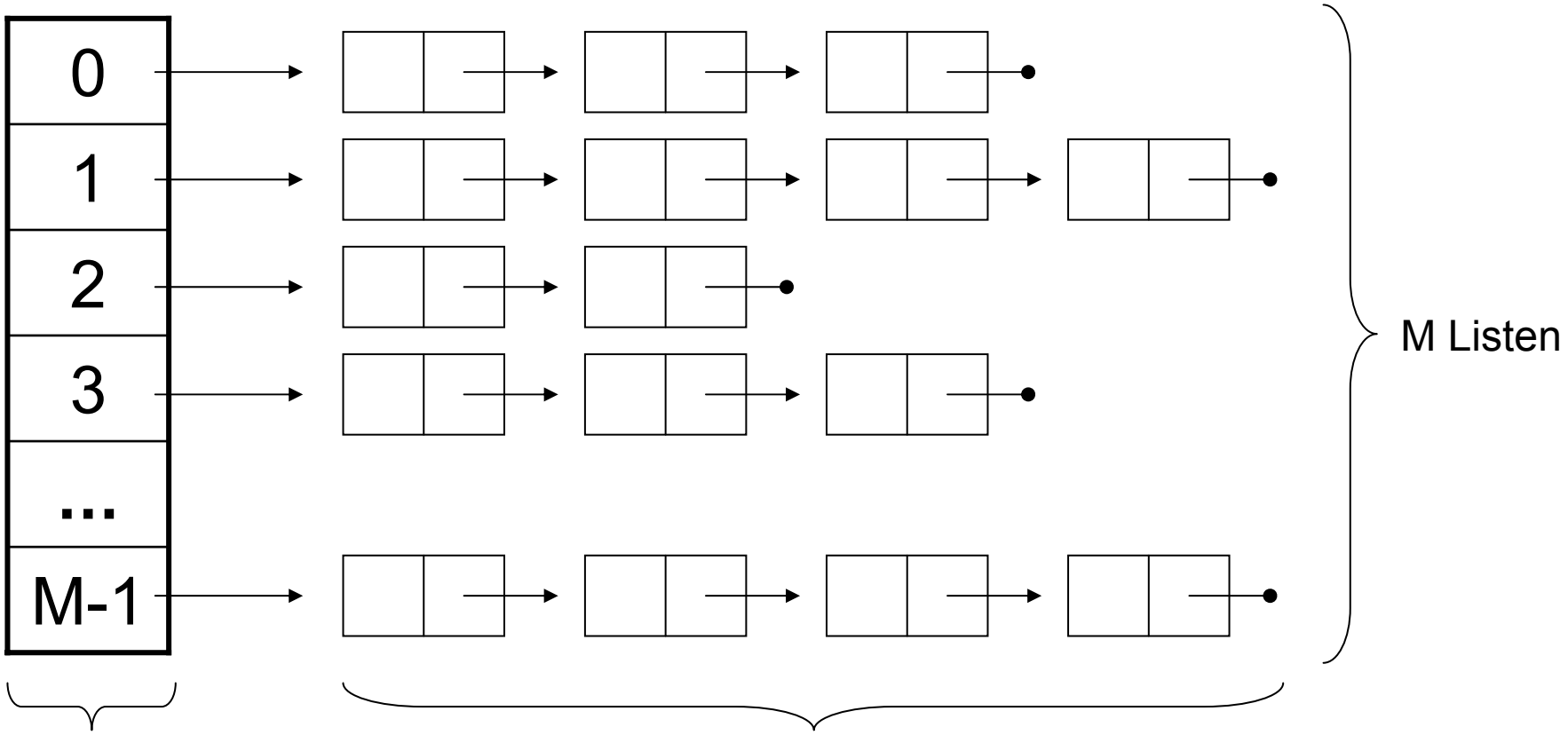
- **Löschen** von Element e

  → Suche nach e in Liste für Hash-Werte h(e)
  Wenn e in der Liste **gefunden** wird, dann aus der Liste entfernen

Auch denkbar: **Ausnahme werfen**, falls

einzufügendes Element schon existiert oder zu löschendes Element nicht vorhanden

Grobentwurf



Was ist zu tun?

1. Wähle Datentyp für die Nutzinformation eines Elements

   ⇒ **hier:** integer (damit der Blick frei für das Wesentliche bleibt)

2. Realisiere den ADT **`IntList`** zur Verarbeitung der Teilmengen

   ⇒ Listen kennen und haben wir schon in C; jetzt aber als C++ Klasse!

3. Realisiere den ADT **`HashTable`**

   ⇒ Verwende dazu den ADT **`IntList`** und eine Hash-Funktion

4. Konstruiere eine Hash-Funktion h: E → { 0,1, …, M – 1}

   ⇒ Kritisch! Wg. Annahme, dass h(·) gleichmäßig über Teilmengen verteilt!

```
class IntList {
private:
  int elem;
  IntList *head;
  IntList *next;

  bool Contains(IntList *aList, int aElem);
  IntList *Delete(IntList *aList, int aElem);
  void Delete(IntList *aList);
  void Print(IntList *aList);

public:
  IntList();
  IntList(int aElem);

  void Insert(int aElem);
  bool Contains(int aElem);
  void Delete(int aElem);
  void Print();

  ~IntList();
};
```

ADT IntList

private rekursive Funktionen

öffentliche Methoden, z.T. überladen

ADT IntList

```
IntList::IntList()
  : head(0), next(0), elem(0) {}

IntList::IntList(int aElem)
  : head(0), next(0), elem(aElem) {}
```

Konstruktoren

```
IntList::~IntList() {
  Delete(head);
}
```

Destruktor

```
void IntList::Delete(IntList *aList) {
  if (aList == 0) return;
  Delete(aList->next);
  delete aList;
}
```

private Hilfsfunktion des Destruktors:

löscht Liste rekursiv!

ADT IntList

öffentliche Methode:

```
bool IntList::Contains(int aElem) {
  return Contains(head, aElem);
}
```

private überladene Methode:

```
bool IntList::Contains(IntList *aList, int aElem) {
  if (aList == 0) return false;
  if (aList->elem == aElem) return true;
  return Contains(aList->next, aElem);
}
```

ADT IntList

öffentliche Methode:

```
void IntList::Delete(int aElem) {
  head = Delete(head, aElem);
}
```

private überladene Methode:

```
IntList *IntList::Delete(IntList *aList, int aElem) {
  if (aList == 0) return 0;
  if (aList->elem == aElem) {
    IntList *tmp = aList->next;  // Zeiger retten
    delete aList;                // Objekt löschen
    return tmp;                  // Zeiger zurückgeben
  }
  aList->next = Delete(aList->next, aElem);
  return aList;
}
```

ADT IntList

öffentliche Methode:

```
void IntList::Print() {
  Print(head);
}
```

private überladene Methode:

```
void IntList::Print(IntList *aList) {
  static int cnt = 1;      // counter
  if (aList != 0) {
    cout << aList->elem;
    cout << (cnt++ % 6 ? "\t" : "\n");
    Print(aList->next);
  }
  else {
    cnt = 1;
    cout << "(end of list)" << endl;
  }
}
```

← Speicherklasse **`static`** : Speicher wird nur einmal angelegt

ADT IntList

öffentliche Methode:

```
void IntList::Insert(int aElem) {
  if (Contains(aElem)) return;
  IntList *newList = new IntList(aElem);
  newList->next = head;
  head = newList;
}
```

→ Implementierung von ADT **`IntList`** abgeschlossen.

ADT HashTable

```
class HashTable {
private:
  IntList **table;
  int maxBucket;
public:
  HashTable(int aMaxBucket);
  int Hash(int aElem) { return aElem % maxBucket; }

  bool Contains(int aElem) {
    return table[Hash(aElem)]->Contains(aElem); }

  void Delete(int aElem) {
    return table[Hash(aElem)]->Delete(aElem); }

  void Insert(int aElem) {
    return table[Hash(aElem)]->Insert(aElem); }

  void Print();
  ~HashTable();
};
```

ADT HashTable

```
HashTable::HashTable(int aMaxBucket) : maxBucket(aMaxBucket) {
  if (maxBucket < 2) throw "invalid bucket size";
  table = new IntList* [maxBucket];
  for (int i = 0; i < maxBucket; i++)
    table[i] = new IntList();
}

HashTable::~HashTable() {
  for (int i = 0; i < maxBucket; i++) delete table[i];
  delete[] table;
}

void HashTable::Print() {
  for (int i = 0; i < maxBucket; i++) {
    cout << "\nBucket " << i << " :\n";
    table[i]->Print();
  }
}
```

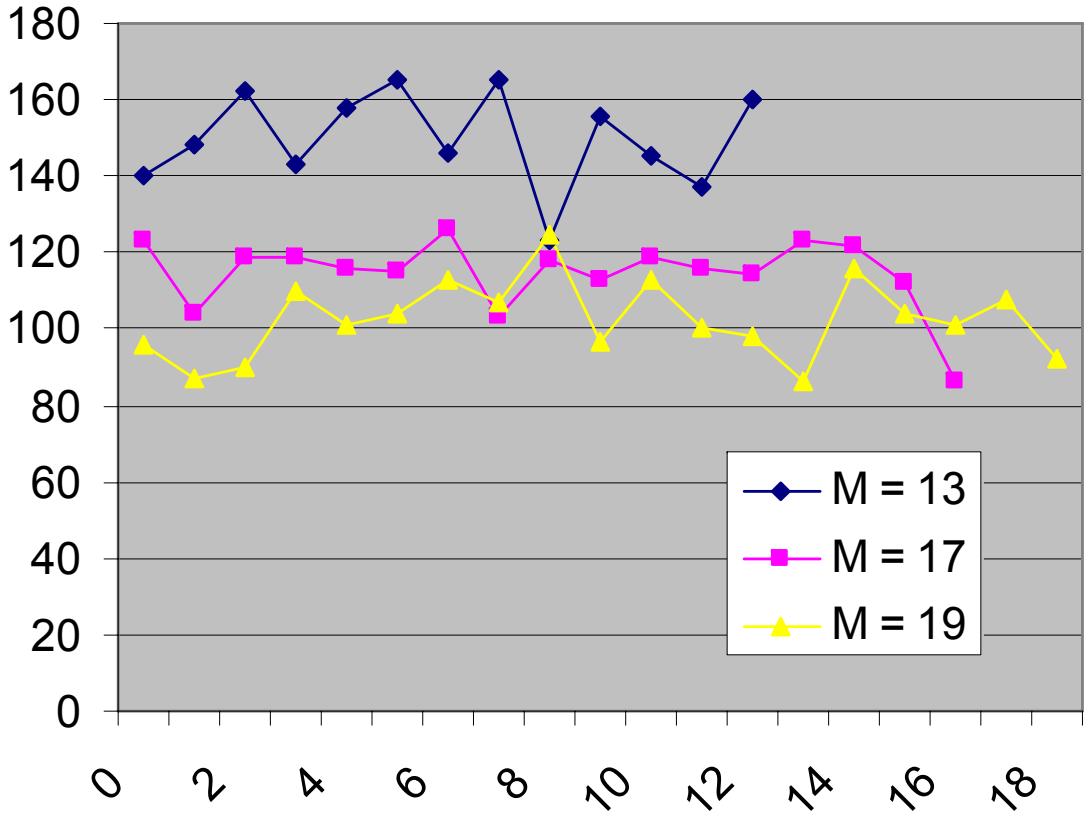
ADT HashTable

```
int main() {
  int maxBucket = 17;
  HashTable *ht = new HashTable(maxBucket);
  for (int i = 0; i < 2000; i++) ht->Insert(rand());

  int hits = 0;
  for (int i = 0; i < 2000; i++)
    if (ht->Contains(rand())) hits++;

  cout << "Treffer: " << hits << endl;
}
```

Ausgabe: **`Treffer: 137`**

`unsigned int`
Pseudozufallszahlen

ADT HashTable: Verteilung von 2000 Zahlen auf M Buckets



M	Mittel-wert	Std.-Abw.
13	149	13,8
17	114	8,1
19	102	6,7

$\Rightarrow$ Hash-Funktion ist wohl OK

Refactoring

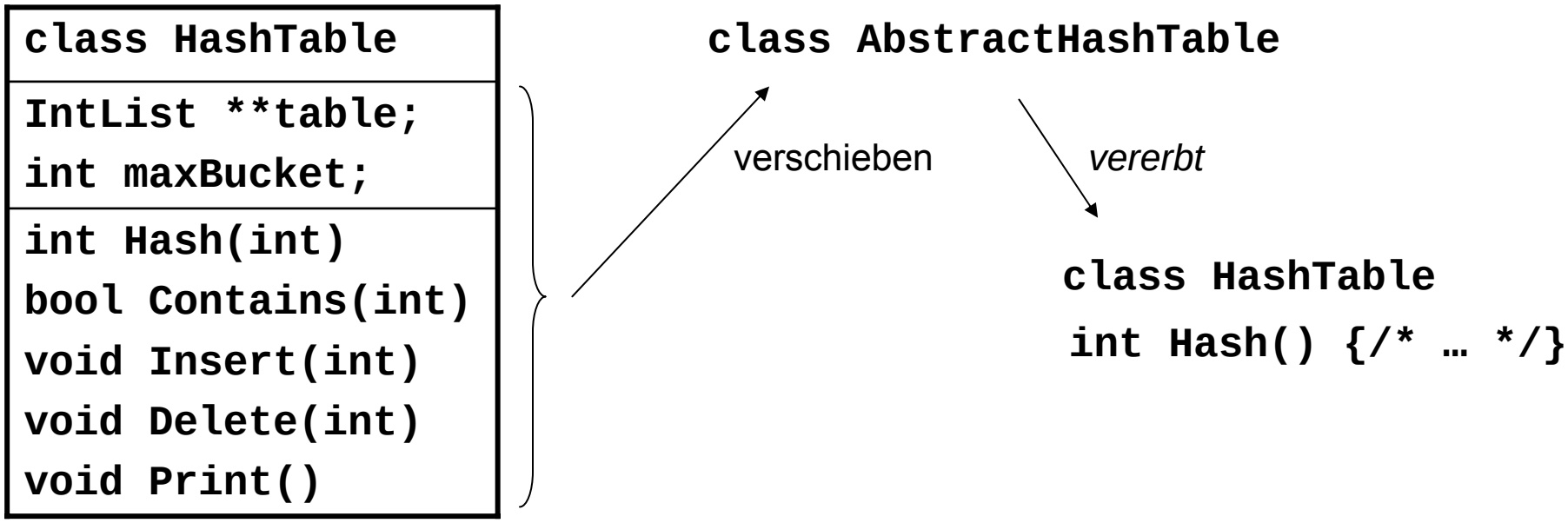
→ Ändern eines Programmteils in kleinen Schritten

- so dass funktionierender abhängiger Code lauffähig bleibt,
- so dass die Gefahr des „Einschleppens“ neuer Fehlern gering ist,
- so dass Strukturen offen gelegt werden, um Erweiterbarkeit zu fördern

```
class HashTable
IntList **table;
int maxBucket;
int Hash(int)
bool Contains(int)
void Insert(int)
void Delete(int)
void Print()
```

Erweiterbarkeit?

Refactoring



1. Neue Basisklasse **`AbstractHashTable`** definieren
2. Attribute und Methoden wandern in Basisklasse
3. **`int Hash(int)`** wird rein virtuell (→ abstrakte Klasse)
   ⇒ **`Hash`** muss in Klasse **`HashTable`** implementiert werden

Refactoring

```
class AbstractHashTable {
private:
  IntList **table;
protected:
  int maxBucket;
public:
  AbstractHashTable(int aMaxBucket);
  virtual int Hash(int aElem) = 0;
  bool Contains(int aElem);
  void Delete(int aElem);
  void Insert(int aElem);
  void Print();
  ~AbstractHashTable();
};
```

⇒ Konsequenzen:

- Code, der HashTable verwendet, kann unverändert bleiben
- Erweiterbarkeit: neue Klassen können von Basisklasse ableiten

```
class HashTable : public AbstractHashTable {
public:
  HashTable(int aMaxBucket) : AbstractHashTable(aMaxBucket) {}
  int Hash(int aElem) { return aElem % maxBucket; }
};
```

Refactoring

```
class HashTable : public AbstractHashTable {
public:
  HashTable(int aMaxBucket) : AbstractHashTable(aMaxBucket) {}
  int Hash(int aElem) { return aElem % maxBucket; }
};
```

```
class HashTable1 : public AbstractHashTable {
public:
  HashTable1(int aMaxBucket) : AbstractHashTable(aMaxBucket){}
  int Hash(int aElem) { return (aElem * aElem) % maxBucket; }
};
```

→ 2 Tests:

(a) Das „alte“ Testprogramm sollte noch funktionieren mit gleicher Ausgabe

(b) Wie wirkt sich neue Hashfunktion von **`HashTable1`** aus?

Refactoring: Test (a)

```
int main() {
  int maxBucket = 17;
  HashTable *ht = new HashTable(maxBucket);
  for (int i = 0; i < 2000; i++) ht->Insert(rand());

  int hits = 0;
  for (int i = 0; i < 2000; i++)
    if (ht->Contains(rand())) hits++;

  cout << "Treffer: " << hits << endl;
}
```

Ausgabe: **`Treffer: 137`**

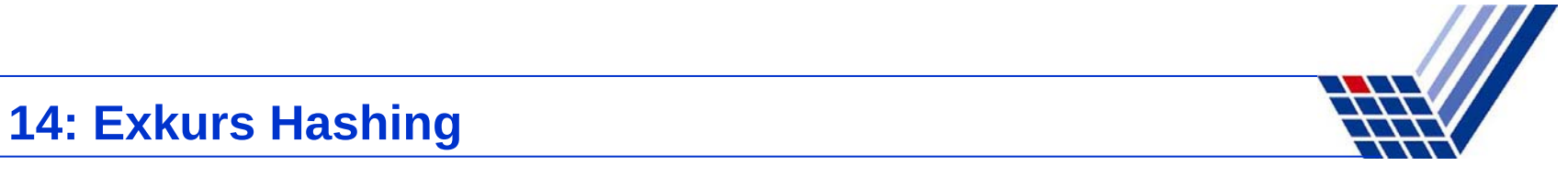
⇒ **Test (a) bestanden!** ☑

Refactoring: Test (b)

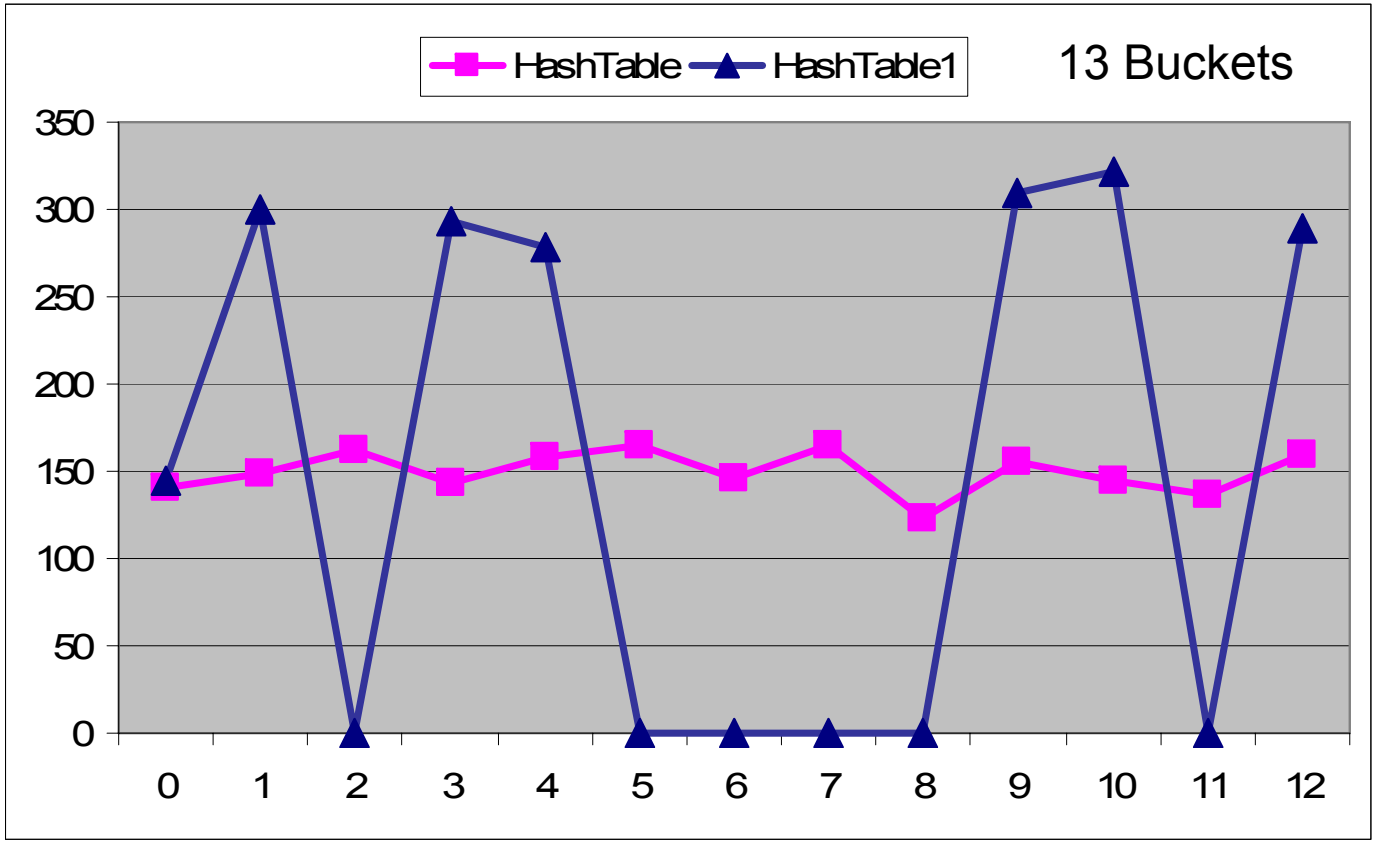
```
int main() {
  int maxBucket = 17;
  HashTable1 *ht = new HashTable1(maxBucket);
  for (int i = 0; i < 2000; i++) ht->Insert(rand());

  int hits = 0;
  for (int i = 0; i < 2000; i++)
    if (ht->Contains(rand())) hits++;

  cout << "Treffer: " << hits << endl;
}
```

Ausgabe: **`Treffer: 137`** ☑

OK, aber wie gleichmäßig verteilt die Hashfunktion die Elemente auf die Buckets?



Refactoring: Test (b)



$\Rightarrow$ Gestalt der Hashfunktion ist von Bedeutung für Listenlängen!