technische universität dortmund

# Einführung in die Programmierung

**Wintersemester 2010/11**

Prof. Dr. Günter Rudolph
Lehrstuhl für Algorithm Engineering
Fakultät für Informatik
TU Dortmund



**Inhalt**

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





## Motivation

**Gesucht:** Datenstruktur zum Einfügen, Löschen und Auffinden von Elementen

⇒ 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

⇒ 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)

⇒ Alternative Suchverfahren notwendig! ⇒ ***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 ∈ E bekommt einen **numerischen** „Stempel“ h(e), der sich aus dem **Dateninhalt** von e berechnet

   Funktion h: E → { 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 Liste),

   Tabelle der Größe M enthält für jeden Hash-Wert eine Liste

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

   Suche nach e → Berechne h(e); h(e) ist Index für Tabelle[ h(e) ] (vom Typ Liste)

   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 Ende 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

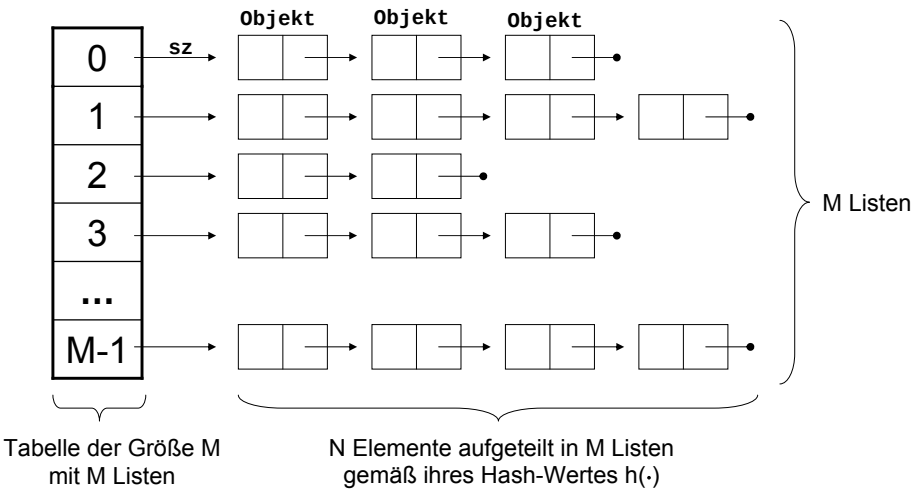
Auch denkbar: **Ausnahme werfen**, falls

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 **`Liste`** zur Verarbeitung der Teilmengen
   ⇒ Listen kennen und haben wir schon; jetzt nur ein paar Erweiterungen!
3. Realisiere den ADT **`HashTable`**
   ⇒ Verwende dazu den ADT **`Liste`** 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 Liste {
public:
  Liste();
  Liste(const Liste& liste);
  void append(const T& x);
  void prepend(const T& x);
  bool empty();
  bool is_elem(const T& x);
  void clear();
  void remove(const T& x);
  void print();
  ~Liste();
private:
  struct Objekt {
    T      data;
    Objekt *next;
  } *sz, *ez;
  void clear(Objekt *obj);
  Objekt* remove(Objekt *obj, const T& x);
  void print(Objekt *obj);
};
```

typedef int T;

ADT Liste

öffentliche Methoden, z.T. überladen

privater lokaler Datentyp

private rekursive Funktionen





### ADT Liste

```
Liste::Liste() : sz(0), ez(0) {
}
```

Konstruktor

```
Liste::~Liste() {
  clear();
}
```

Destruktor

```
void Liste::clear() {
  clear(sz);
  sz = ez = 0;
}
```

***`public clear`*** : gibt Speicher frei, initialisiert zu leerer Liste

```
void Liste::clear(Objekt *obj) {
  if (obj == 0) return;
  clear(obj->next);
  delete obj;
}
```

private Hilfsfunktion von ***`public clear`***

löscht Liste rekursiv!





### ADT Liste

öffentliche Methode:

```
void Liste::remove(const T& x){
  sz = remove(sz, x);
}
```

private überladene Methode:

```
Liste::Objekt* Liste::remove(Objekt *obj, const T& x) {
  if (obj == NULL) return NULL;      // oder: Ausnahme!
  if (obj->data == x) {
    Objekt *tmp = obj->next;         // Zeiger retten
    delete obj;                      // Objekt löschen
    return tmp;                      // Zeiger retour
  }
  obj->next = remove(obj->next, x); // Rekursion
  if (obj->next == NULL) ez = obj;
  return obj;
}
```

## ADT Liste

öffentliche Methode:

```
void Liste::print() {
  print(sz);
}
```

private überladene Methode:

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

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





## ADT HashTable

```
class HashTable {
private:
  Liste *table;
  unsigned int maxBucket;
public:
  HashTable(int aMaxBucket);
  int Hash(int aElem) { return aElem % maxBucket; }
  bool Contains(int aElem) {
    return table[Hash(aElem)].is_elem(aElem); }
  void Delete(int aElem) {
    table[Hash(aElem)].remove(aElem); }
  void Insert(int aElem) {
    table[Hash(aElem)].append(aElem); }
  void Print();
  ~HashTable();
};
```





## ADT HashTable

```
HashTable::HashTable(int aMaxBucket) : maxBucket(aMaxBucket) {
  if (maxBucket < 2) throw "invalid bucket size";
  table = new Liste[maxBucket];
}

HashTable::~HashTable() {
  delete[] table;
}

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





## ADT HashTable

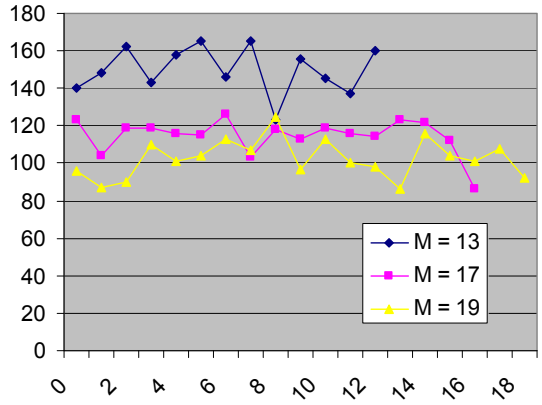
```
int main() {
  unsigned int maxBucket = 17;
  HashTable ht(maxBucket);
  for (int i = 0; i < 2000; i++) ht.Insert(rand());

  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

**Achtung!** Das Ergebnis erhält man nur unter Verwendung der schlecht realisierten Bibliotheksfunktion rand() von MS Windows. Unter Linux: 0.

**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 |

⇒ Hash-Funktion ist wohl OK





**Noch ein Test ...**

```
int main() {
  unsigned int maxBucket = 17;
  HashTable ht(maxBucket);
  T a[500];
  int k = 0;
  for (unsigned int i = 0; i < 2000; i++) {
    unsigned int x = rand();
    ht.Insert(x);
    if (k < 500 && ht.Hash(x) < 3) a[k++] = x;
  }
  while (--k >= 0) {
    if (!ht.Contains(a[k])) cerr << a[k] << endl;
    ht.Delete(a[k]);
  }
  ht.Print();
  return 0;
}
```





**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
Liste *table;
int maxBucket;
int Hash(int)
bool Contains(int)
void Insert(int)
void Delete(int)
void Print()
```

→ Erweiterbarkeit?





**Refactoring**

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

verschieben

```
class AbstractHashTable
```

*vererbt*

```
class HashTable
int Hash() {/* … */}
```

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:
  Liste *table;
protected:
  int maxBucket;
public:
  AbstractHashTable(int aMaxBucket);
  virtual int Hash(T aElem) = 0;
  bool Contains(T aElem);
  void Delete(T aElem);
  void Insert(T 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(T aElem) { return aElem % maxBucket; }
};
```





## Refactoring

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

```
class HashTable1 : public AbstractHashTable {
public:
  HashTable1(int aMaxBucket) : AbstractHashTable(aMaxBucket){}
  int Hash(T 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!** ☑





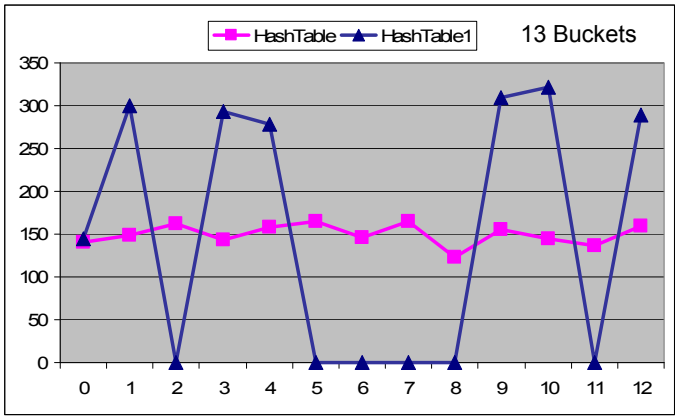
## Refactoring: Test (b)

```
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)




⇒ Gestalt der Hashfunktion ist von Bedeutung für Listenlängen!