

Einführung in die Programmierung

Wintersemester 2018/19

Prof. Dr. Günter Rudolph

Lehrstuhl für Algorithm Engineering

Fakultät für Informatik

TU Dortmund

Inhalt

Hashing

- Motivation
- Grobentwurf
- ADT Liste (ergänzen)
- ADT HashTable
- Anwendung

Mergesort

- Konzept
- Laufzeitanalyse
- Realisierung (mit Schablonen)

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} , $\{A, B, \dots, Z\}$, ...

Partielle Ordnung

Es existieren unvergleichbare Elemente: $a \parallel b$ $\begin{pmatrix} 2 \\ 5 \end{pmatrix} < \begin{pmatrix} 8 \\ 6 \end{pmatrix}$; $\begin{pmatrix} 2 \\ 5 \end{pmatrix} \parallel \begin{pmatrix} 3 \\ 4 \end{pmatrix}$
Beispiele: \mathbb{N}^2 , \mathbb{R}^3 ...

Idee: durch lexikographische Ordnung total machen! **Aber:** Degenerierte Bäume!

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

Funktion $h: E \rightarrow \{ 0, 1, \dots, M - 1 \}$ 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 $e \rightarrow$ Berechne $h(e)$; $h(e)$ ist Index für $\text{Tabelle}[h(e)]$ (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

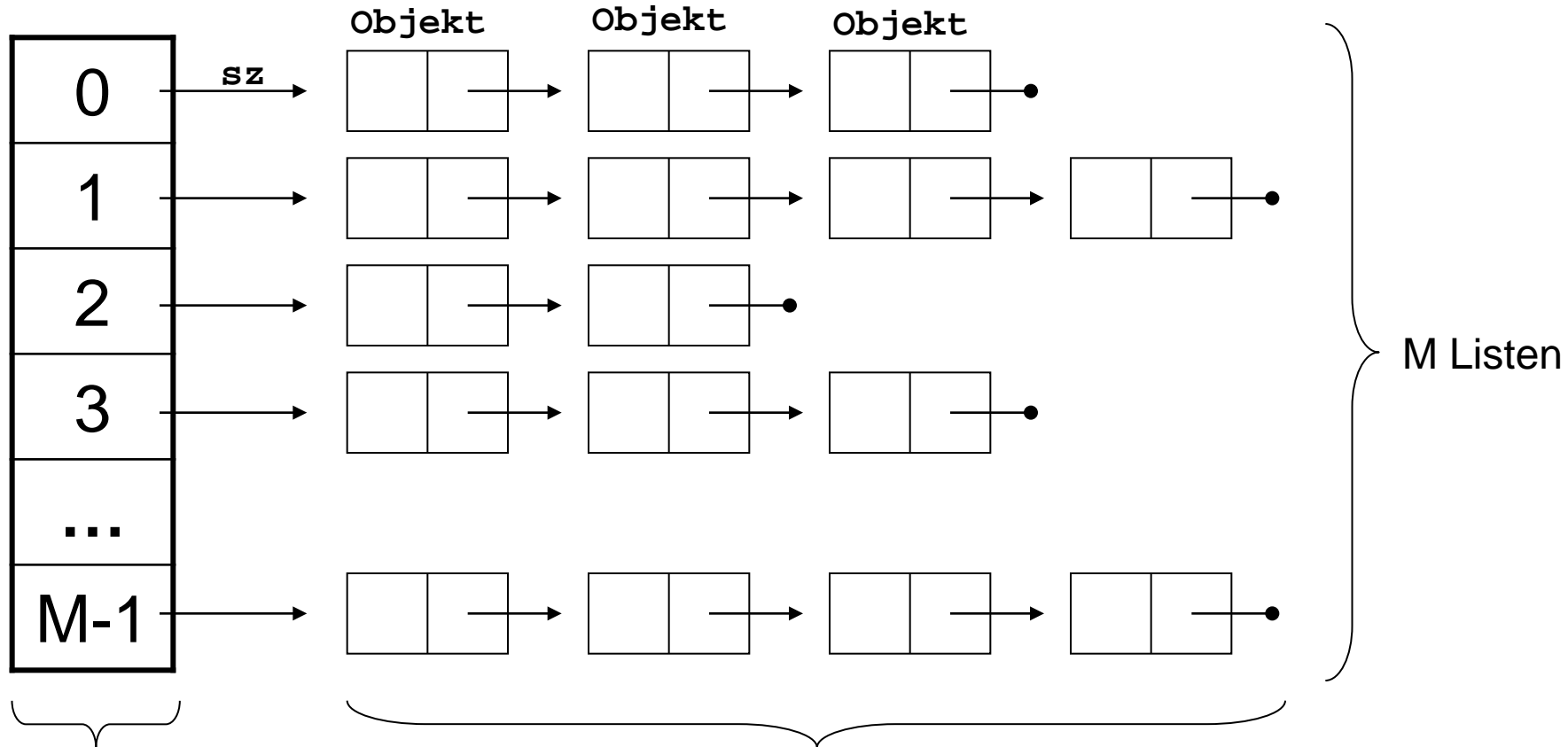


Tabelle der Größe M
mit M Listen

N Elemente aufgeteilt in M Listen
gemäß ihres Hash-Wertes $h(\cdot)$

Was ist zu tun?

1. Wähle Datentyp für die Nutzinformation eines Elements
⇒ **hier:** realisiert als Schablone!
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 $h: E \rightarrow \{0, 1, \dots, M - 1\}$
⇒ Kritisch! Wg. Annahme, dass $h(\cdot)$ gleichmäßig über Teilmengen verteilt!

```

template<typename T> 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();
protected:
    struct Objekt {
        T      data;
        Objekt *next;
    } *sz, *ez;
    void clear(Objekt *obj);
    Objekt* remove(Objekt *obj, const T& x);
    void print(Objekt *obj);
};
    
```

ADT Liste

öffentliche
Methoden,
z.T. überladen

privater lokaler
Datentyp

private rekursive
Funktionen

ADT Liste

```
template<typename T> Liste<T>::Liste()
: sz(nullptr), ez(nullptr) {
}
```

Konstruktor

```
template<typename T> Liste<T>::~~Liste() {
    clear();
}
```

Destruktor

```
template<class T> void Liste<T>::clear() {
    clear(sz);
    sz = ez = nullptr;
}
```

public clear :
gibt Speicher frei,
initialisiert zu leerer
Liste

```
template<typename T>
void Liste<T>::clear(Objekt *obj) {
    if (obj == nullptr) return;
    clear(obj->next);
    delete obj;
}
```

private Hilfsfunktion
von *public clear*

löscht Liste rekursiv!

ADT Liste

öffentliche Methode:

```
template<typename T> void Liste<T>::remove(const T& x){
    sz = remove(sz, x); if(sz == nullptr) ez = nullptr;
}
```

private überladene Methode:

```
template<typename T>
Liste<T>::Objekt* Liste<T>::remove(Objekt *obj, const T& x) {
    if (obj == nullptr) return nullptr; // 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 == nullptr) ez = obj;
    return obj;
}
```

ADT Liste

öffentliche Methode:

```
template<typename T> void Liste::print() {  
    print(sz);  
}
```

private überladene Methode:

```
template<typename T>  
void Liste::print(Objekt *obj) {  
    static int cnt = 1;    // counter  
    if (obj != nullptr) {  
        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

```
template<typename T> class HashTable {
private:
    Liste<T> *table;
    unsigned int maxBucket;
public:
    HashTable(int aMaxBucket);
    virtual int Hash(T& aElem) = 0;           // rein virtuell!
    bool Contains(T& aElem) {
        return table[Hash(aElem)].is_elem(aElem); }
    void Delete(T& aElem) {
        table[Hash(aElem)].remove(aElem); }
    void Insert(T& aElem) {
        table[Hash(aElem)].append(aElem); }
    void Print();
    ~HashTable();
};
```

ADT HashTable

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

template<typename T>
HashTable<T>::~~HashTable() {
    delete[] table;
}

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

ADT HashTableInt

```
class HashTableInt : public HashTable<int> {  
public:  
    int Hash(int& aElem) { return aElem % maxBucket; }  
}
```


ADT HashTable

```
int main() {
    unsigned int maxBucket = 17;
    HashTableInt 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;
    return 0;
}
```

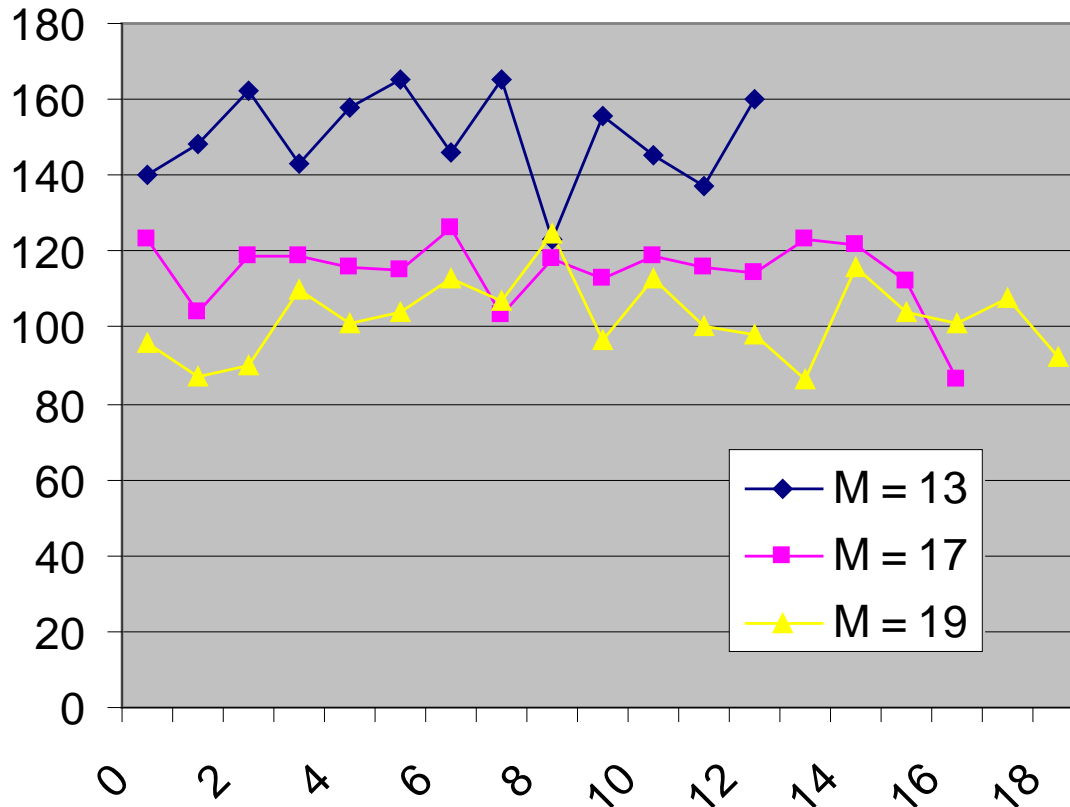
Ausgabe: Treffer: 137

unsigned int
Pseudozufallszahlen



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

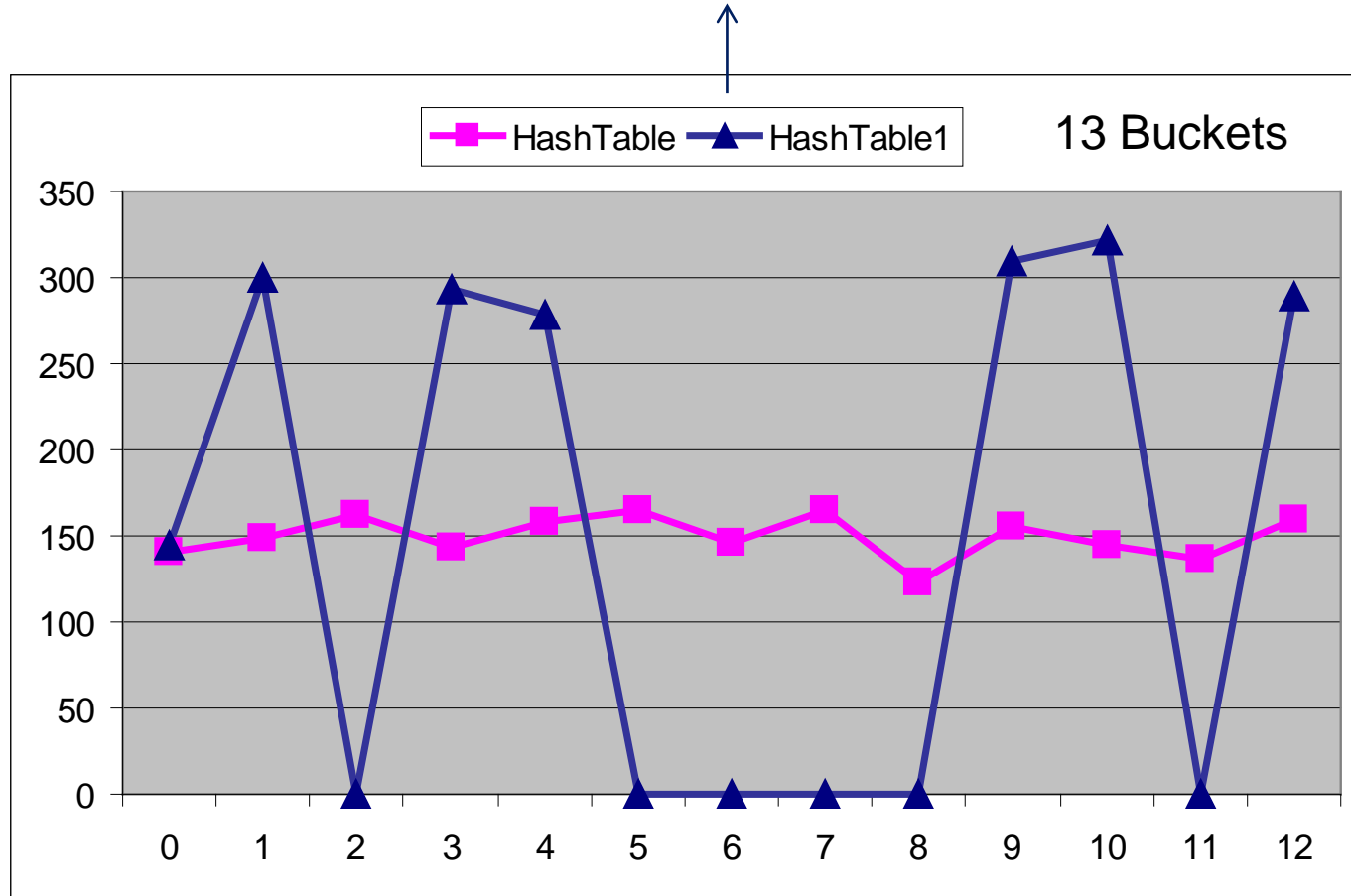
ADT HashTable: Verteilung von 2000 Zahlen auf M Buckets



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

⇒ Hash-Funktion ist wohl OK

```
int Hash(T aElem) { return (aElem * aElem) % maxBucket; }
```



⇒ Gestalt der Hashfunktion ist von Bedeutung für Listenlängen!

Graphische Anwendung: Vektoren $(x_1, x_2, x_3) \in [a, b] \subset \mathbb{N}^3$ wiederfinden

$$H(x) = \left[C \cdot \sum_{i=1}^3 \lambda_i \cdot (x_i - a_i) \right] \bmod M$$

wobei $\lambda_i > 0$ und $C = \frac{2^{32} - 1}{\sum_{i=1}^3 \lambda_i \cdot (b_i - a_i)}$
 $\left. \begin{array}{l} \} \\ \} \end{array} \right\}$

max. Zahlenbereich

max. Summenwert

Falls $M = 2^k$ für $(k < 32)$,

dann Modulo-Operation schnell durch logisches AND berechenbar:

$$\text{Hash} = \text{floor}(C * \text{sum}(x)) \& (M-1)$$

Aufgabe: Texte wiederfinden

Problem: ungleichmäßige Verteilung von Worten und Buchstabengruppen

⇒ alle n Zeichen der Zeichenkette x einbeziehen

$$H_i = \begin{cases} x_1 & , \text{ falls } i = 1 \\ (\lambda \cdot H_{i-1} + x_i) \bmod M & , \text{ falls } i > 1 \end{cases}$$

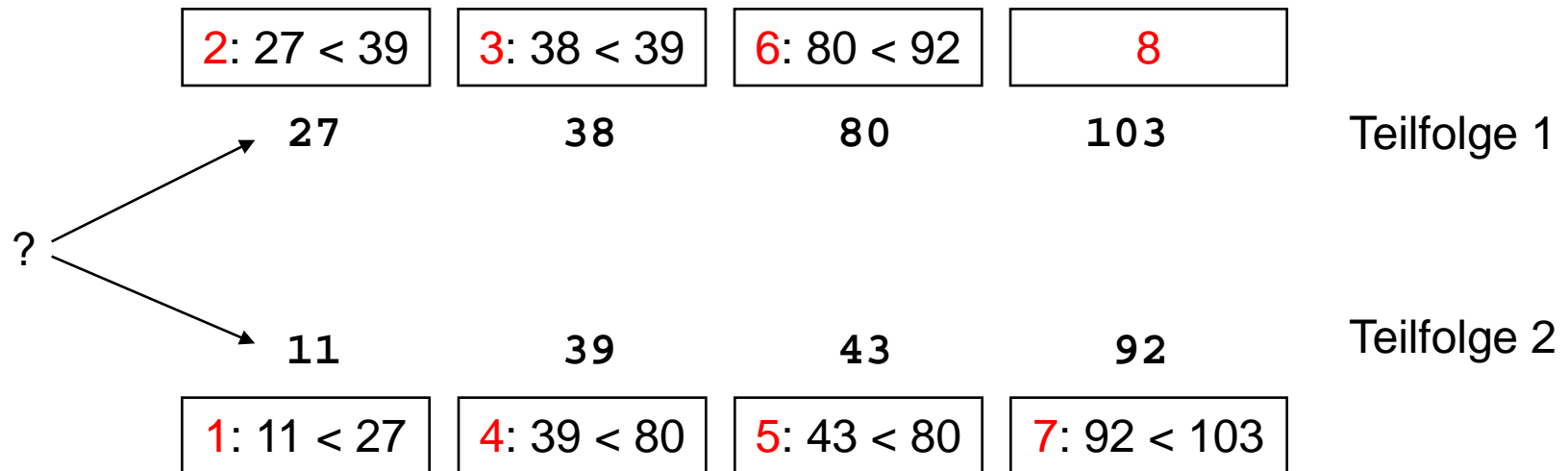
„Rolling Hash“

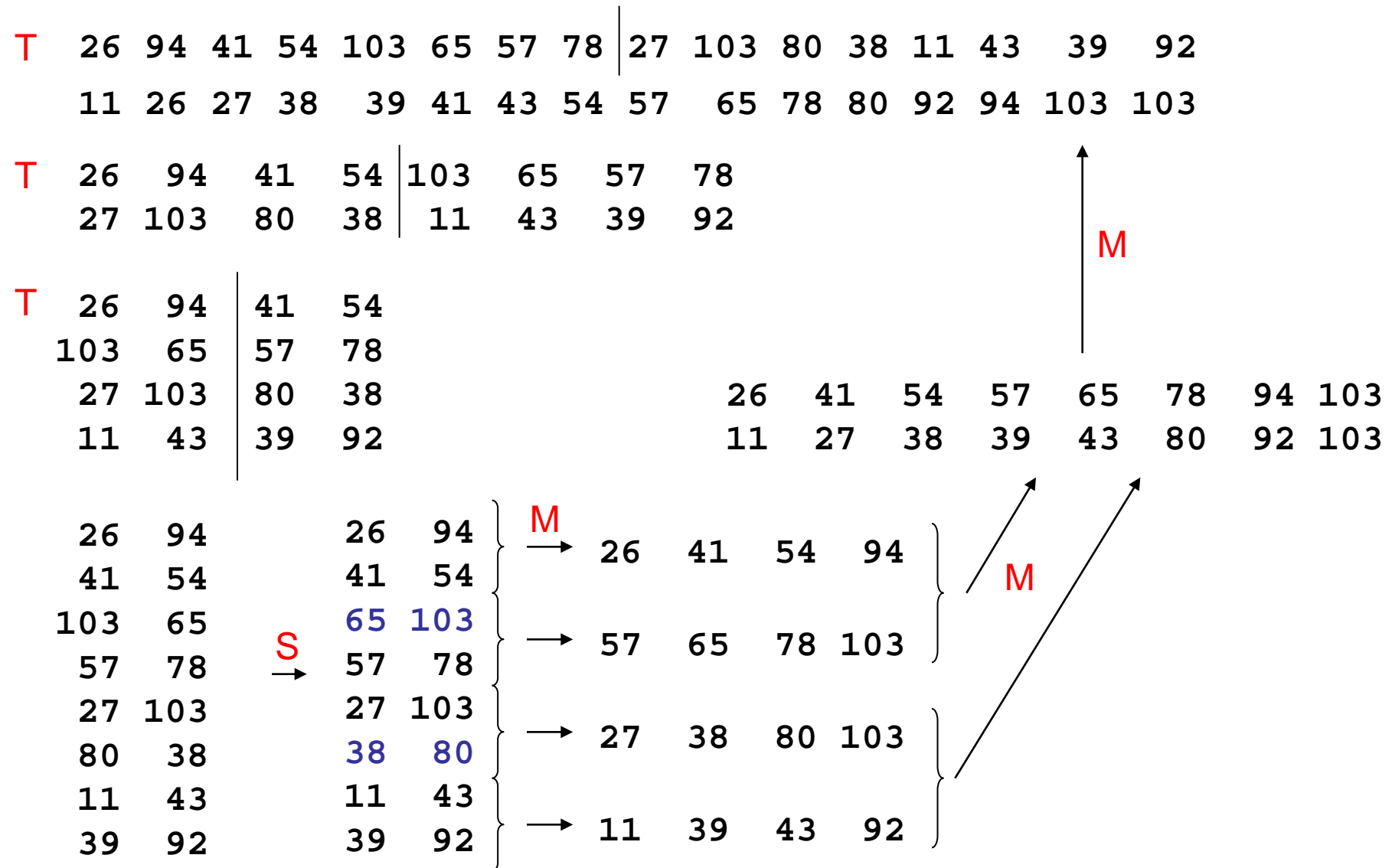
H_n ist der Hashwert der Zeichenkette x mit n Zeichen

Mergesort

Beobachtung:

Sortieren ist einfach, wenn man zwei sortierte Teilfolgen hat.





Laufzeitanalyse

Annahme: Anzahl Objekte $n = 2^k \Leftrightarrow k = \log_2 n$

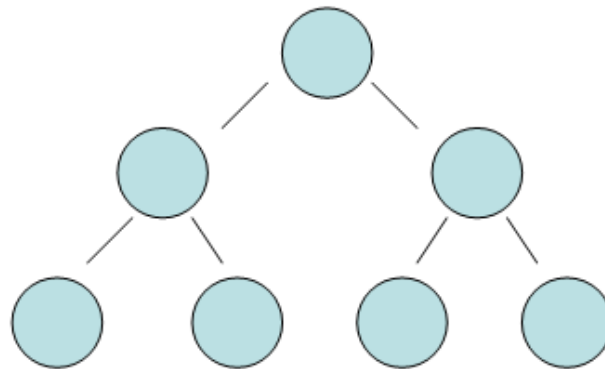
2^0 Teilsequenzen

2^1 Teilsequenzen

2^2 Teilsequenzen

⋮

2^{k-1} Teilsequenzen



2^k Objekte je Teilsequenz

2^{k-1} Objekte je Teilsequenz

2^{k-2} Objekte je Teilsequenz

⋮

$2^{k-(k-1)}$ Objekte je Teilsequenz
 = 2

(a) 2^{k-1} Vergleiche zum Sortieren der 2^{k-1} Paare

(b) auf Ebene e: $(2^{k-e}-1)$ Vergleiche zum Mischen von 2 der 2^e Sequenzen

$\Rightarrow (2^{k-e}-1) \star 2^{e-1} = 2^{k-1} - 2^{e-1}$ Vergleiche auf Ebene $e = 1, \dots, k-1$

$\Rightarrow 2^{k-1} + (k-1) \star 2^{k-1} - \text{Summe}(2^{e-1}; 1..k-1) = (k-1) \star 2^{k-1} + 1 < k \star 2^k = n \log_2 n$

Mergesort

- Eingabe: unsortiertes Feld von Zahlen
- Ausgabe: sortiertes Feld
- Algorithmisches Konzept: „Teile und herrsche“ (*divide and conquer*)
 - Zerlege Problem solange in Teilprobleme bis Teilprobleme lösbar
 - Löse Teilprobleme
 - Füge Teilprobleme zur Gesamtlösung zusammen

Hier:

1. Zerteile Feld in Teilfelder bis Teilproblem lösbar (→ bis Feldgröße = 2)
2. Sortiere Felder der Größe 2 (→ einfacher Vergleich zweier Zahlen)
3. Füge sortierte Teilfelder durch Mischen zu sortierten Feldern zusammen

Mergesort

- Programmentwurf

1. Teilen eines Feldes → einfach!

2. Sortieren

- a) eines Feldes der Größe 2 → einfach!

- b) eines Feldes der Größe > 2 → rekursiv durch Teilen & Mischen

3. Mischen → nicht schwer!

Annahme:

Feldgröße ist
Potenz von 2

Mergesort: Version 1

```
void Msort(int const size, int a[]) {
    if (size == 2) { // sortieren
        if (a[0] > a[1]) Swap(a[0], a[1]);
        return;
    }
    // teilen
    int k = size / 2;
    Msort(k, &a[0]);
    Msort(k, &a[k]);
    // mischen
    Merge(k, &a[0], &a[k]);
}
```

} sortieren (einfach)

} sortieren durch Teilen
& Mischen

```
void Swap(int& a, int& b) {
    int c = b; b = a; a = c;
}
```

} Werte vertauschen
per Referenz

Mergesort: Version 1

```
void Merge(int const size, int a[], int b[]) {
    int* c = new int[2*size];

    // mischen
    int i = 0, j = 0;
    for (int k = 0; k < 2 * size; k++)
        if ((j == size) || (i < size && a[i] < b[j]))
            c[k] = a[i++];
        else
            c[k] = b[j++];

    // umkopieren
    for (int k = 0; k < size; k++) {
        a[k] = c[k];
        b[k] = c[k+size];
    }
    delete[] c;
}
```

← dynamischen
Speicher
anfordern

dynamischen
Speicher
freigeben
←

Mergesort: Version 1

```
void Print(int const size, int a[]) {
    for (int i = 0; i < size; i++) {
        cout << a[i] << "\t";
        if ((i+1) % 8 == 0) cout << endl;
    }
    cout << endl;
}

int main() {
    int const size = 32;
    int a[size];

    for (int k = 0; k < size; k++) a[k] = rand();

    Print(size, a);
    Msort(size, a);
    Print(size, a);

    return 0;
}
```

Hilfsfunktion
für
Testprogramm

Programm
zum Testen

Mergesort: Version 1

Ausgabe:

41	18467	6334	26500	19169	15724	11478	29358
26962	24464	5705	28145	23281	16827	9961	491
2995	11942	4827	5436	32391	14604	3902	153
292	12382	17421	18716	19718	19895	5447	21726
41	153	292	491	2995	3902	4827	5436
5447	5705	6334	9961	11478	11942	12382	14604
15724	16827	17421	18467	18716	19169	19718	19895
21726	23281	24464	26500	26962	28145	29358	32391

OK, funktioniert für `int` ... was ist mit `char`, `float`, `double` ... ?

⇒ **Idee:** Schablonen!

Mergesort: Version 2

```
template <class T> void Msort(int const size, T a[]) {
    if (size == 2) { // sortieren
        if (a[0] > a[1]) Swap<T>(a[0], a[1]);
        return;
    }
    // teilen
    int k = size / 2;
    Msort<T>(k, &a[0]);
    Msort<T>(k, &a[k]);
    // mischen
    Merge<T>(k, &a[0], &a[k]);
}
```

```
template <class T> void Swap(T& a, T& b) {
    T c = b; b = a; a = c;
}
```

Mergesort: Version 2

```
template <class T> void Merge(int const size, T a[], T b[]) {
    T* c = new T[2*size];

    // mischen
    int i = 0, j = 0;
    for (int k = 0; k < 2 * size; k++) {
        if ((j == size) || (i < size && a[i] < b[j]))
            c[k] = a[i++];
        else
            c[k] = b[j++];

        // umkopieren
        for (int k = 0; k < size; k++) {
            a[k] = c[k];
            b[k] = c[k+size];
        }
    }
    delete[] c;
}
```


Mergesort: Version 2

```
template <class T> void Print(int const size, T a[]) { ... }
```

```
int main() {  
    int const size = 32;  
  
    int a[size];  
    for (int k = 0; k < size; k++) a[k] = rand();  
    Print<int>(size, a);  
    Msort<int>(size, a);  
    Print<int>(size, a);  
  
    float b[size];  
    for (int k = 0; k < size; k++) b[k] = rand() * 0.01f;  
    Print<float>(size, b);  
    Msort<float>(size, b);  
    Print<float>(size, b);  
  
    return 0;  
}
```

↑
Konstante
vom Typ float
(nicht double)

Mergesort: Version 2

Ausgabe:

41	18467	6334	26500	19169	15724	11478	29358
26962	24464	5705	28145	23281	16827	9961	491
2995	11942	4827	5436	32391	14604	3902	153
292	12382	17421	18716	19718	19895	5447	21726

41	153	292	491	2995	3902	4827	5436
5447	5705	6334	9961	11478	11942	12382	14604
15724	16827	17421	18467	18716	19169	19718	19895
21726	23281	24464	26500	26962	28145	29358	32391

147.71	115.38	18.69	199.12	256.67	262.99	170.35	98.94
287.03	238.11	313.22	303.33	176.73	46.64	151.41	77.11
282.53	68.68	255.47	276.44	326.62	327.57	200.37	128.59
87.23	97.41	275.29	7.78	123.16	30.35	221.9	18.42

7.78	18.42	18.69	30.35	46.64	68.68	77.11	87.23
97.41	98.94	115.38	123.16	128.59	147.71	151.41	170.35
176.73	199.12	200.37	221.9	238.11	255.47	256.67	262.99
275.29	276.44	282.53	287.03	303.33	313.22	326.62	327.57

Mergesort: Version 2

Schablone instantiiert mit Typ `string` funktioniert auch!

Schablone instantiiert mit Typ `Complex` funktioniert **nicht!** Warum?

Vergleichsoperatoren sind nicht überladen für Typ `Complex`!

in `Msort`: `if (a[0] > a[1]) Swap<T>(a[0], a[1]);`

in `Merge`: `if ((j == size) || (i < size && a[i] < b[j]))`

Entweder Operatoren überladen oder überladene Hilfsfunktion (z.B. `Less`):

```
bool Less(Complex &x, Complex &y) {  
    if (x.Re() < y.Re()) return true;  
    return (x.Re() == y.Re() && x.Im() < y.Im());  
}
```

hier:
lexikographische
Ordnung