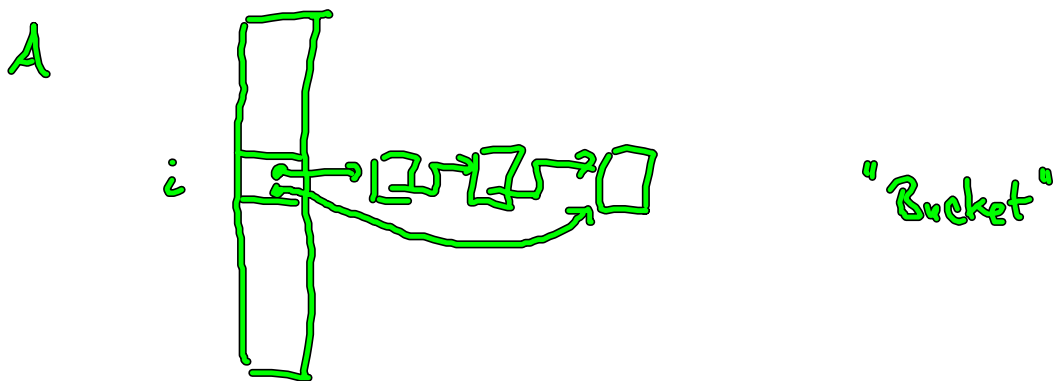


# Bucket sort

## einfacher Bucket sort

- kennen Schlüsselmenge  $m$  Schlüssel
- Richte Array von Wertschlangen ein  
( Listen mit Einfügen am Ende )



- Durchlaufe die  $n$  Objekte
- hat das Objekt Schlüssel  $i$ , so füge es in die Wertschlange bei  $A[i]$  ein
- Am Ende konkateniere die nicht leeren Listen  $A[0], A[1], \dots, A[m-1]$

$$O(n + m)$$

Anwendung auf Sortierung von Strings (lexikographisch)  
voraussetz: alle Strings haben Länge  $k$

einf. Bucketsort der Komponenten, hier an gefangene  
Liste am Ende sortiert die Strings lexikographisch

Invariante

Nach  $i$  Durchläufen von Bucketsort  
sind die Strings nach den letzten  
 $i$  Komponenten lexikogr. sortiert

Aufwand:  $k$  Durchläufe

einfacher Bucketsort pro Durchlauf

$$\Rightarrow \Theta(k(n+m)) = O(k \cdot n)$$

↑  
falls  $m \leq n$

spezielle Anwendung:

Sortieren von  $n$   $k$ -stelligen Dualzahlen  $O(k \cdot n)$

Comp I:  $O(n \log n)$  zum Sortieren von  $n$  Zahlen

? Widerspruch ?

Kein Widerspruch

Falls  $n$  paarw. versch. Dualzahlen brauchen zur  
Darstellung eine Stellenzahl  $k \geq \log n$

$\underbrace{\quad\quad\quad}_k \rightarrow 2^k$  Zahlen darst.  
 $2^k \geq n$

Jetzt Strings unterschiedlicher Länge mit Bucket sort sortieren

erste Idee: alle Strings gleicher Länge, links anfangen und Strings, die dort kein Zeichen haben besonders behandeln (Bucket = kein Zeichen)

$\Rightarrow O(l_{max} \cdot (k + m))$   
 $\uparrow$   
 max. Länge eines Strings

Bsp: bab   abc   a   \_

Buckets   kein Zeichen a   }  
           a  
           b    b a b  
           c    a b c   }  $\rightarrow$  a \_ bab   abc

kein Zeichen   a   }  
                   a    b a b  
                   b  
                   c    a b c   }  $\rightarrow$  a   bab   abc

kein Zeile

a  
b  
c

a  
bab

abc

} →

a abc bab

geht besser in  $O(l_{\text{total}} + m)$

↑

Gesamtzahl der Zeichen



Idee ① in einem Durchgang wo die Strings nehmen die dort ein Zeichen haben

[ Durchgang  $\hat{=}$  Stelle  $i$

$\Rightarrow$  Stringlänge  $\geq i$  ]

② vorab nötige Buckets für jeden Durchgang ermitteln  
macht Konkrete schneller

zu ① Durchlaufe alle Strings, füge sie in  
Liste LENGTH [i] ein

LENGTH ist array von Listen

an Komp.  $i$  stehen alle Strings mit

Länge  $i$

Bsp.    bab    abc    a                     $\rightarrow$  ?  
           3        3        1                     $O(l_{total})$   
 LENGTH [1]    a  
           [2]     $\emptyset$   
           [3]    bab abc

## zu ② Anwendung von Bucketsort

- Durchlaufe alle Strings  
 erzeuge Paare  $(i, s)$  wenn ein String Zeichen  $s$   
 an Position  $i$  hat

bab abc a                     $O(l_{total})$   
   ↓    ↘    ↗  
 (1,b) (2,a) (3,b) (1,a) (2,b) (3,c) (1,a)  
           ↙                     $O(l_{total} + m + l_{total} + l_{max})$

- Sortiere diese Paare lexikogr. mit  $l_{i,s}$  max Stellenzahl  
 für Strings gleicher Länge

(1,a) (1,a) (1,b) (2,a) (2,b) (3,b) (3,c)

- Verwende Array NONEMPTY  
 NONEMPTY[i] enthält sortierte Liste  
 der Zeichen, die an Position  $i$  vorkommen  
 [ Diese Zeichen  $\hat{=}$  nicht-leere Buckets  
 bei Bucketsort und  $i$ -te Stelle ]

NONEMPTY [1]    a, b                     $O(l_{total})$   
                   [2]    a, b

[3] b,c

## Verwendung von ① und ②

Durchlaufe Stellen  $i$  von hinten nach vorn  
pro Stelle behandle nur die bisherige Liste  
(aus vorigem Durchlauf)

⊗ vorangestellt durch  $LENGTH[i]$

Kalkuliere und Durchlauf die Buckets  
die ein Zeichen in  $NONEMPTY[i]$  haben

Bsp: bab abc a

LENGTH [1] a

[2]

[3] bab abc

NONEMPTY [1] a, b

[2] a, b

[3] b, c

Stelle  $i=3$       LENGTH [3]    bab    abc

Bucket [a]

[b] bab

[c] abc

→ nur diese kerkat.  
bab abc

Stelle  $i=2$       bab    abc

Bucket [a]    bab

[b]    abc

[c]

← zur kerkat  
bab abc

Stelle  $i = 1$

a    bab    abc

LENGTH [1]

Bucket [a]	a	abc	}	a	abc	bab
[b]	bab					
[c]						

⊗ vorausgesetzt wichtig um Invariante zu erhalten

Nach Bucketart für Position  $i$

sind alle Strings, die ein Zeichen bei  $i$  haben  
und den Stellen  $i, i+1, \dots$  lexikogr.

korrekt sortiert

⇒ Verfahren arbeitet korrekt

Aufwand für Preprocessing in ① und ②

$$O(l_{\text{total}}) + O(l_{\text{total}}) + O(l_{\text{total}} + m + l_{\text{total}} + l_{\text{max}}) + O(l_{\text{total}})$$

$$= O(l_{\text{total}} + m + l_{\text{max}}) = O(l_{\text{total}} + m)$$

$$\uparrow \\ l_{\text{max}} \leq l_{\text{total}}$$

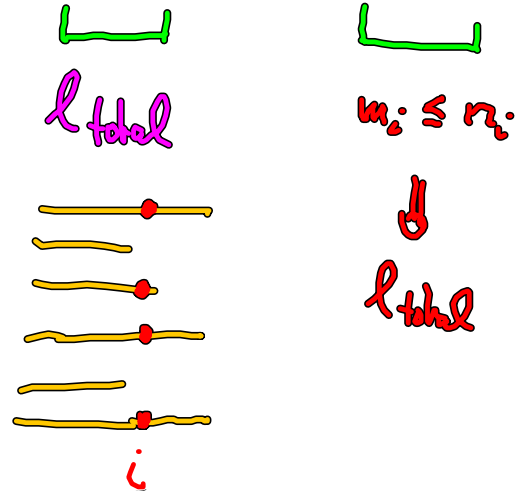
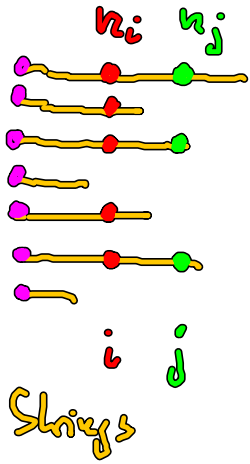
Aufwand für das Sortieren

Im Durchlauf für Stelle  $i$      $n_i$  Strings

$l_{max}$  Stellen

$m_i$  Buckets

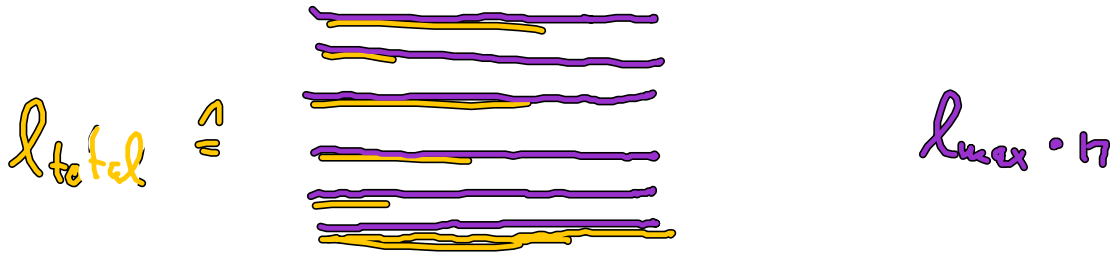
$$\sum_{i=1}^{l_{max}} O(n_i + m_i) = O\left(\sum_{i=1}^{l_{max}} n_i\right) + O\left(\sum_{i=1}^{l_{max}} m_i\right)$$



$\Rightarrow$  Sortie run :  $O(l_{total}) + O(l_{total}) = O(l_{total})$

insgesamt :  $O(l_{total} + m)$

zum Vergleich zu  $O(l_{max} (n + m))$



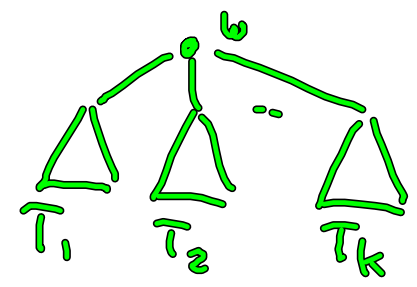
§ 16 Bäume

# rekursive Def

# Def als gerichtete Graphen

siehe Lemma I

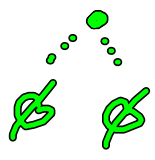
- leerer Baum ist Baum
- Gegeben Bäume  $T_1, \dots, T_k$  mit Wurzeln  $w_1, \dots, w_k$  dann ist auch  $T$  ges. durch neue Wurzel  $w$  mit Teilbäumen  $T_1 \dots T_k$  ein Baum



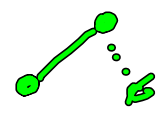
## binäre Bäume

$k \leq 2$

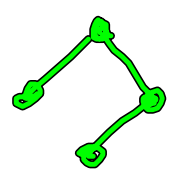
$\emptyset$



=  $\bullet$



=  $\bullet - \bullet$

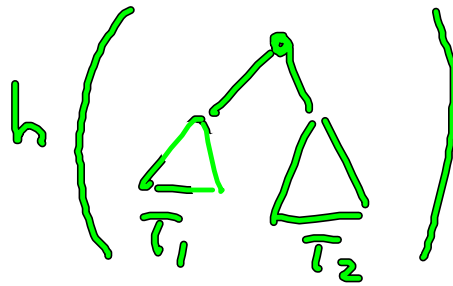


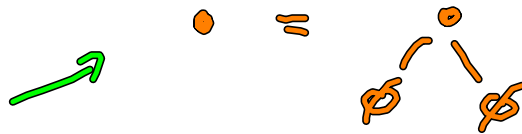
Kindo, Sohne, Vater, ... Blatt, interne Knoten, ..

rekursive Struktur wichtig in Algorithmen und in charakteristischen Größen von Bäumen

# Höhe eines binären Baumes (rekursiv)

$$h(\emptyset) := -1$$

$$h \left( \begin{array}{c} \text{ } \\ \diagup \quad \diagdown \\ \text{ } \end{array} \right) := \max \{ h(T_1), h(T_2) \} + 1$$


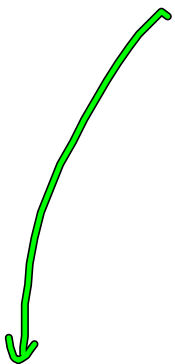
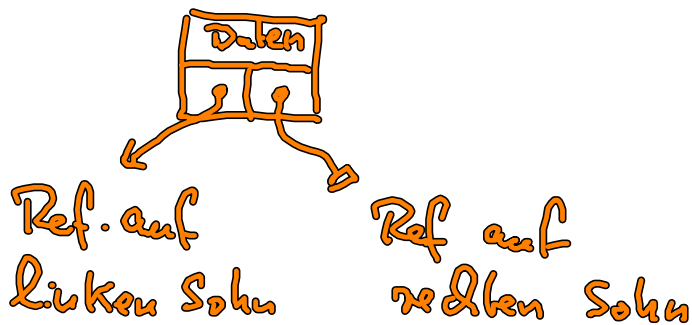


$$\text{Höhe } 0 = \max \{ -1, -1 \} + 1$$

## Klasse für binäre Bäume

```
class Tree {  
    TreeNode root;  
    Tree getLTree();  
    Tree getRTree();  
    int getHeight();  
    ...  
}
```

```
class TreeNode {  
    Object data;  
    TreeNode lson;  
    TreeNode rson;  
    ...  
}
```



```
int getHeight() {  
    if (root == NULL) return -1;  
    int lh = this.getLeftTree().getHeight();  
    int rh = this.getRightTree().getHeight();  
    return Math.max(lh, rh) + 1;  
}
```