TECHNISCHE UNIVERSITÄT DRESDEN

ALGORITHMEN UND DATENSTRUKTUREN

ÜBUNG 9: SORTIEREN, SUCHEN & KORRIGIEREN

Eric Kunze
eric.kunze@mailbox.tu-dresden.de

TU Dresden, 19.12.2019

DARUM GEHT'S HEUTE

```
void (*(*f[])())()
```

```
f                   -- f
f[]                 -- is an array
*f[]                -- of pointers ([] has higher precede
(*f[])()            -- to functions
*(*f[])()           -- returning pointers
(*(*f[])())()       -- to functions
void (*(*f[])())(); -- returning void
```

```
void (*(*f[])())()
```

```
f                     -- f
f[]                   -- is an array
*f[]                  -- of pointers ([] has higher precede
(*f[])()              -- to functions
*(*f[])()             -- returning pointers
(*(*f[])())()         -- to functions
void (*(*f[])())();   -- returning void
```

...NICHT ☺

QUICKSORT

```
void quicksort(int a[], int L, int R) {
    int i, j, w, x, k;

    i = L;  j = R;  k = (L + R) / 2;
    x = a[k];

    do {
        while (a[i] < x) i = i + 1;
        while (a[j] > x) j = j - 1;
        if (i <= j) {
            w = a[i];     //
            a[i] = a[j];  //  swap a[i] and a[j]
            a[j] = w;     //
            i = i + 1;  j = j - 1;
        }
    } while (i <= j);

    if (L < j) quicksort(a, L, j);
    if (R > i) quicksort(a, i, R);
}
```

AUFGABE 1

1. Durchlauf

$$\begin{array}{ccccc}
9 & 13 & \boxed{7} & 6 & 10 \\
\uparrow & & & & \uparrow \\
i & & & & j \\
9 & 13 & \boxed{7} & 6 & 10 \\
\uparrow & & & \uparrow & \\
i & & & j & \\
6 & 13 & \boxed{7} & 9 & 10 \\
 & \uparrow & \uparrow & & \\
 & i & j & & \\
6 & \boxed{7} & 13 & 9 & 10 \\
 & \uparrow & \uparrow & & \\
 & j & i & &
\end{array}$$

2. Durchlauf

$$\begin{array}{ccc||ccc}
 & \boxed{6} & 7 & 13 & \boxed{9} & 10 \\
 & \uparrow & \uparrow & \uparrow & & \uparrow \\
 & i & j & i & & j \\
 & \boxed{6} & 7 & 13 & \boxed{9} & 10 \\
 & \uparrow & & \uparrow & \uparrow & \\
 & i,j & & i & j & \\
 & \boxed{6} & 7 & \boxed{9} & 13 & 10 \\
\uparrow & & \uparrow & \uparrow & \uparrow & \\
j & & i & j & i &
\end{array}$$

3. Durchlauf

$$
\begin{array}{cc||cc||cc}
6 & 7 & 9 & & \boxed{13} & 10 \\
 & & & & \uparrow & \uparrow \\
 & & & & i & j \\
6 & 7 & 9 & & 10 & \boxed{13} \\
 & & & \uparrow & & \uparrow \\
 & & & j & & i
\end{array}
$$

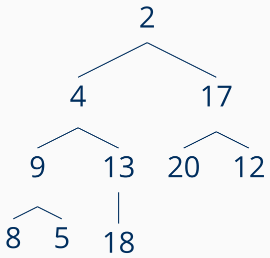
Ergebnis:

`[6,7,9,10,13]`

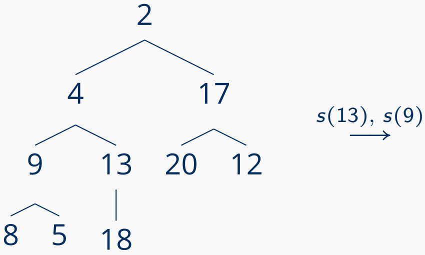
HEAPSORT

- Bäume sind nur Veranschaulichung
- Algorithmus arbeitet auf Listen
- zwei Phasen
  - **1. Phase:** Einsortieren in den Heap und Herstellen der Heap-Eigenschaft
  - **2. Phase:** Führe Sortierschritt wiederholt durch:
    - Tausch von Wurzel und ”letztem“ Element (tiefste Ebene, ganz rechts)
    - Fixiere dieses Element
    - Sinkenlassen des neuen Wurzelelements

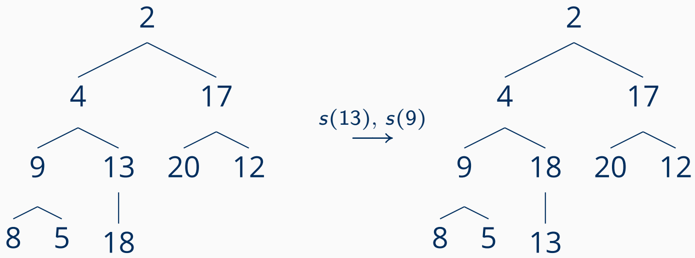
AUFGABE 2 — PHASE 1



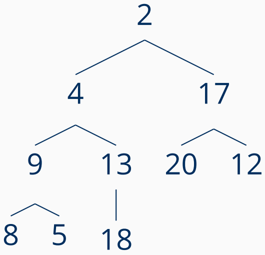
AUFGABE 2 — PHASE 1



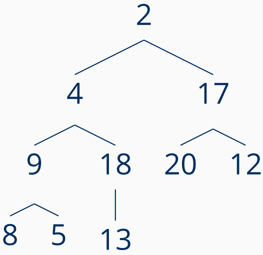
AUFGABE 2 — PHASE 1



AUFGABE 2 — PHASE 1

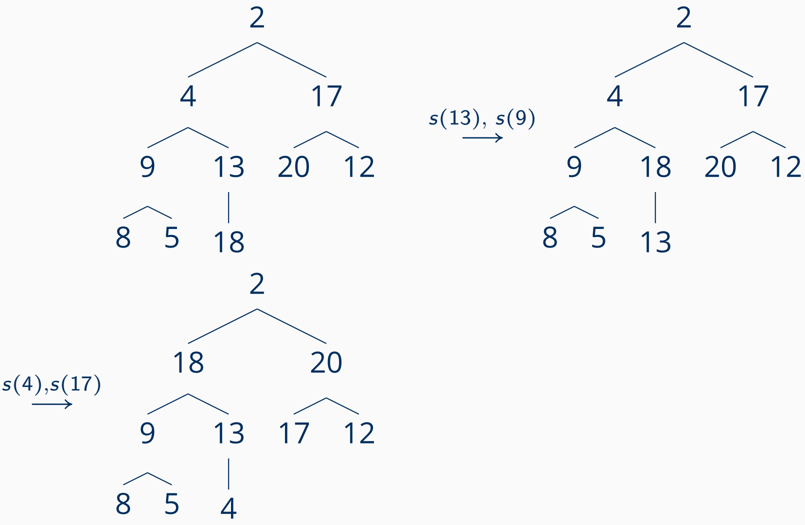


$\xrightarrow{s(13),\ s(9)}$

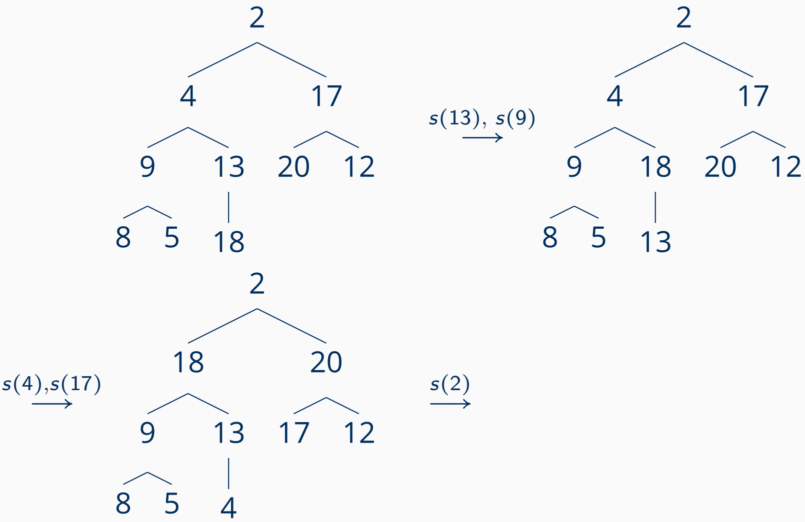


$\xrightarrow{s(4),s(17)}$

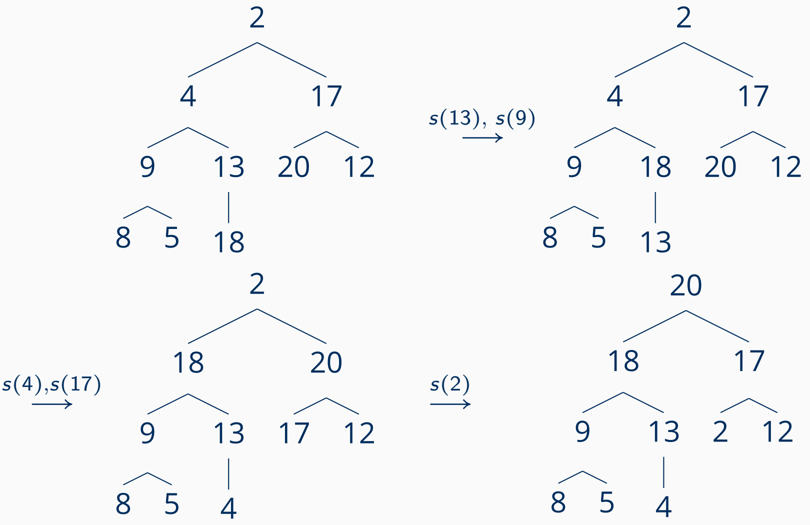
AUFGABE 2 — PHASE 1



AUFGABE 2 — PHASE 1



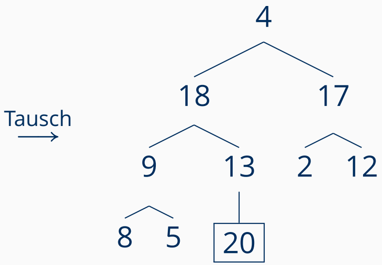
AUFGABE 2 — PHASE 1



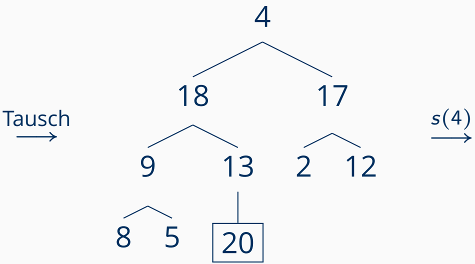
AUFGABE 2 — PHASE 2

$\xrightarrow{\text{Tausch}}$

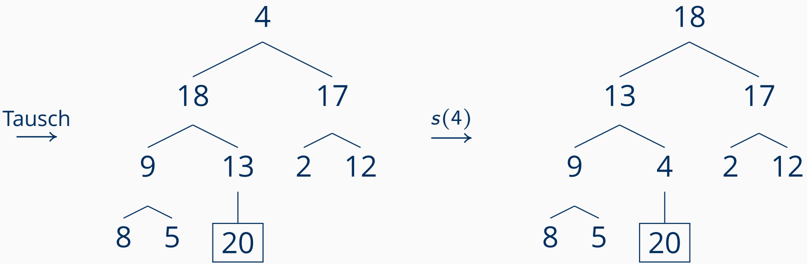
AUFGABE 2 — PHASE 2



AUFGABE 2 — PHASE 2

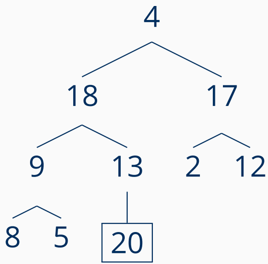


AUFGABE 2 — PHASE 2

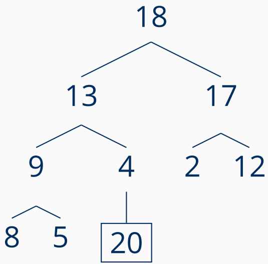


AUFGABE 2 — PHASE 2

$\xrightarrow{\text{Tausch}}$

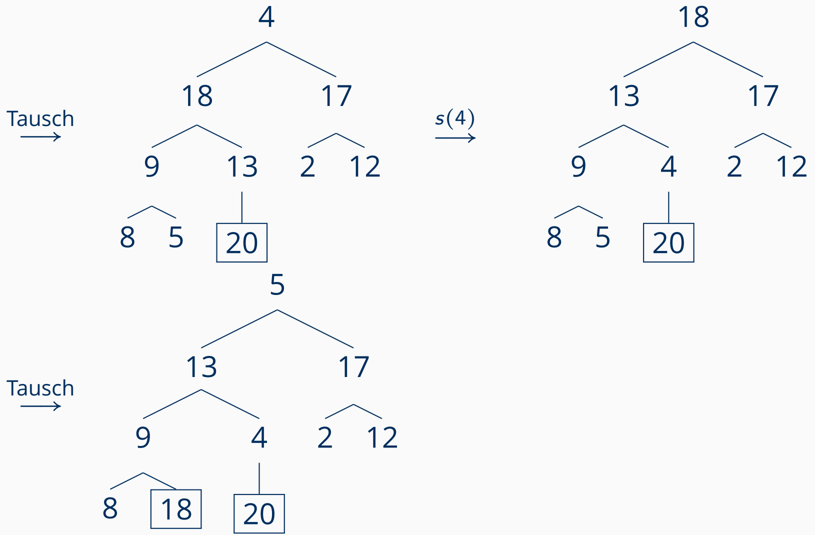


$\xrightarrow{s(4)}$

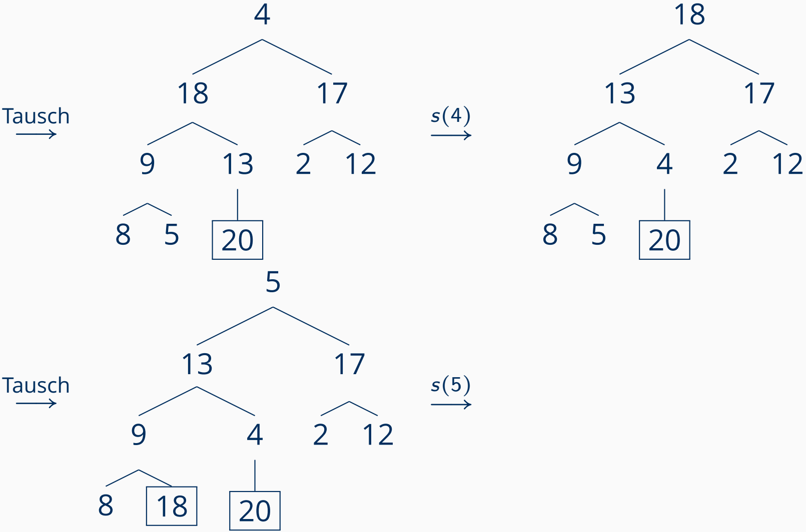


$\xrightarrow{\text{Tausch}}$

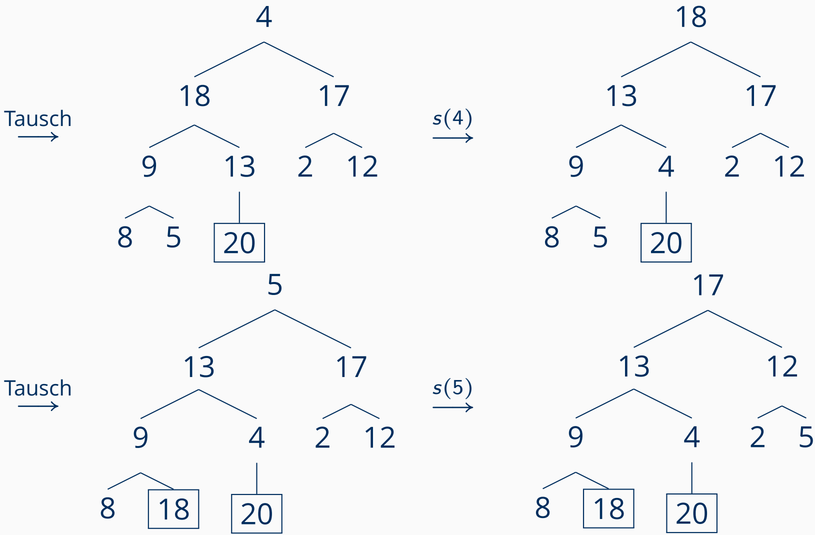
AUFGABE 2 — PHASE 2



AUFGABE 2 — PHASE 2



AUFGABE 2 — PHASE 2



KMP-ALGORITHMUS

- Mustersuche in (großen) Texten
- Ziel: Verschiebung des Musters um mehr als eine Position bei Nichtübereinstimmung.
- Methode: Ermittlung einer Verschiebetabelle `Tab[]`
- Bedeutung des Eintrags `Tab[i]=j`:
  Bei Nichtübereinstimmung an Stelle i wird Position j des Musters an aktueller Vergleichsstelle angelegt.

KMP-ALGORITHMUS

Suche das Muster `aaabaaaa` im Text `aaabaaabaaacaaabaaaa`.

Position	0	1	2	3	4	5	6	7
Pattern	a	a	a	b	a	a	a	a
Tabelle	-1	-1	-1	2	-1	-1	-1	3

Erster Versuch:

a a a b a a a **b** a a a c a a a b a a a a
a a a b a a a **a**

Tabelleneintrag an Position 7 ist 3, d.h. `Tab[7] = 3` — Lege Position 3 des Musters an aktueller Vergleichsposition an:

a a a b a a a b a a a **c** a a a b a a a a
a a a b a a a **a**

Erneute Verschiebung liefert schließlich:

a a a b a a a b a a a c a a a b a a a a
a a a b a a a a

KMP-ALGORITHMUS — DIE ZYKLENMETHODE

Zwei Phasen:

KMP-ALGORITHMUS — DIE ZYKLENMETHODE

Zwei Phasen:

- **1. Phase:** Markieren der längsten Teilwörter im Pattern, die mit einem Präfix übereinstimmen
  - ein Zyklus beginnt an einer Patternposition `i` falls $i \neq 0$ und `Pat[0] = Pat[i]`
  - ein Zyklus endet an der kleisten Patternposition `i+m`, sodass $\texttt{Pat[m+1]} \neq \texttt{Pat[i+m+1]}$

KMP-ALGORITHMUS — DIE ZYKLENMETHODE

Zwei Phasen:

- **1. Phase:** Markieren der längsten Teilwörter im Pattern, die mit einem Präfix übereinstimmen
  - ein Zyklus beginnt an einer Patternposition `i` falls `i ≠ 0` und `Pat[0] = Pat[i]`
  - ein Zyklus endet an der kleisten Patternposition `i+m`, sodass `Pat[m+1] ≠ Pat[i+m+1]`
- **2. Phase:** Bestimmung der Tabelleneinträge
  - `Tab[0] = -1`
  - Tabelleneinträge nach einem Zyklus: *Länge des längsten dort endenden Zyklus*
  - Tabelleneinträgen in einem Zyklus: *Tabelleneintrag der derzeitigen Position im längsten laufenden Zyklus*
  - verbleibende Einträge: `0`

AUFGABE 3

Teil (a) Pattern: `aabaaacaab`

AUFGABE 3

Teil (a) Pattern: `aabaaacaab`

Position	0	1	2	3	4	5	6	7	8	9
Pattern	a	a	b	a	a	a	c	a	a	b
Tabelle	-1	-1	1	-1	-1	2	2	-1	-1	1

AUFGABE 3

Teil (a) Pattern: `aabaaacaab`

Position	0	1	2	3	4	5	6	7	8	9
Pattern	a	a	b	a	a	a	c	a	a	b
Tabelle	-1	-1	1	-1	-1	2	2	-1	-1	1

Teil (b)

Position	0	1	2	3	4	5
Pattern	*c*	*b*	c	c	b	*a*
Tabelle	-1	0	-1	1	0	2

LEVENSHTEIN-DISTANZ

Kosten zur Überführung eines Wortes $w = w_1 \ldots w_n$ in ein Wort $v = v_1 \ldots v_k$; schreibe $d(w_1 \ldots w_j, v_1 \ldots v_i) = d(j, i)$.

$$d(0, i) = i$$

$$d(j, 0) = j$$

$$d(j, i) = \min\{d(j, i-1) + 1, d(j-1, i) + 1, d(j-1, i-1) + \delta_{j,i}\}$$

für alle $1 \leq j \leq n$ und alle $1 \leq i \leq k$ wobei

$$\delta_{j,i} = \begin{cases} 1 & \text{wenn } w_j \neq v_i \\ 0 & \text{sonst} \end{cases}$$

LEVENSHTEIN-DISTANZ

Kosten zur Überführung eines Wortes $w = w_1 \dots w_n$ in ein Wort $v = v_1 \dots v_k$; schreibe $d(w_1 \dots w_j, v_1 \dots v_i) = d(j, i)$.

$$d(0, i) = i$$

$$d(j, 0) = j$$

$$d(j, i) = \min\{d(j, i-1) + 1, d(j-1, i) + 1, d(j-1, i-1) + \delta_{j,i}\}$$

für alle $1 \le j \le n$ und alle $1 \le i \le k$ wobei

$$\delta_{j,i} = \begin{cases} 1 & \text{wenn } w_j \neq v_i \\ 0 & \text{sonst} \end{cases}$$

Anschaulich: Überlagerung durch Pattern → Pfeile zeigen ”Ursprung“ des Minimums an

$w_j \neq v_i$:

+1	+1
+1	?

$w_j = v_i$:

+0	+1
+1	?

AUFGABE 4

$d(j, i)$			**D**		**i**		**s**		**t**		**a**		**n**		**z**
	0	→	1	→	2	→	3	→	4	→	5	→	6	→	7
	↓	↘													
D	1		0	→	1	→	2	→	3	→	4	→	5	→	6
	↓		↓	↘											
i	2		1		0	→	1	→	2	→	3	→	4	→	5
	↓		↓		↓	↘		↘		↘		↘			
n	3		2		1		1	→	2	→	3		3	→	4
	↓		↓		↓	↘		↘		↘		↘	↓	↘	
s	4		3		2		1	→	2	→	3	→	4		4
	↓		↓		↓		↓	↘							
t	5		4		3		2		1	→	2	→	3	→	4
	↓		↓		↓		↓		↓	↘					
a	6		5		4		3		2		1	→	2	→	3
	↓		↓		↓	↘	↓		↓		↓	↘		↘	
s	7		6		5		4		3		2		2	→	3

AUFGABE 4

Alignments mit minimaler Levenshtein-Distanz:

$$\begin{array}{cccccccc} D & i & n & s & t & a & * & s \\ | & | & | & | & | & | & | & | \\ D & i & * & s & t & a & n & z \\ & & d & & & & i & s \end{array}$$

$$\begin{array}{cccccccc} D & i & n & s & t & a & s & * \\ | & | & | & | & | & | & | & | \\ D & i & * & s & t & a & n & z \\ & & d & & & & s & i \end{array}$$