Datenstrukturen − Kurzanleitung

Insertionsort

p = (72, 25, 4, 83, 6, 60) n = 6

Start: K_1 ist sortierte Folge.

Für i = 2, 3, 4 , ... , n:

Füge k_i in eine sortierte Folge (k'_1, k'_2, k'_3, ... , k'_n) in der richtigen Position ein.

1	2	3	4	5	6	
72 |	25	4	83	6	60	
25	72 |	4	83	6	60	Schritt 1
4	25	72 |	83	6	60	Schritt 2
4	24	72	83 |	6	60	Schritt 3
4	6	25	72	83 |	60	Schritt 4
3	5	2	6	1	4	
4	6	25	60	72 |	83	Schritt5

g_i sind die Anzahl der Elemente, die links von k_i stehen und echt größer als k_i sind.

i	1	2	3	4	5	6
k_i	72	25	4	83	6	60
g_i	0	1	2	0	3	2

$$\sum_{i=1}^{n} g_i = 8$$

Anzahl der Vergleiche:

$$V(p)=\sum_{i=1}^{n} g_i+(n-1) \quad = 8 + (6 - 1) = \underline{13}$$

Anzahl der Bewegungen:

$$B(p)=\sum_{i=1}^{n} g_i+3(n-1) \quad = 8 + 3 (6 - 1) = \underline{23}$$

Insertionsort ist <u>stabil</u> !

Begr: Eine Vertauschung der Elemente findet nur statt, wenn der links stehende Satz echt größer ist. Bei Schlüsselgleichheit bricht die while−Schleife ab.

Shellsort

p = (9, 8, 2, 10, 1, 3, 7, 6, 5, 4) n = 10 h = 3

Als h−Sortierung bezeichnet man die unabhängige Sortierung der h−Teilfolgen.

9	8	2	10	1	3	7	6	5
7			9			10		
	1			6			8	
		2			3			5
7	1	2	9	6	3	10	8	5

Inkrementfolgen $h_1 > h_2 > h_3 > ... > h_t$

a) Knuth 1973 $\frac{3^j-1}{2}$

j	1	2	3	4	5	6	7	8	9	10	....
$\frac{3^j-1}{2}$	1	4	13	46	121	364	1093	3280	9481	29524	....

$h_1 < n/2$

Bsp: n = 250 => n/2 = 125

Inkrementfolge (121, 46, 13, 4, 1)

b) Hibbard 1963 2^j-1

j	1	2	3	4	5	6	7	8	9	10	....
2^j-1	1	3	7	15	31	63	127	255	511	1023	....

$h_1 < n/2$

Bsp: n = 30 => n/2 = 15

Inkrementfolge (7, 3, 1)

c) Gonnet − Folge 1984

α = 0,45454

$h_1 = \alpha * n$ größte ganze Zahl <= $\alpha * n$

$h_2 = \alpha * h_1$

$h_3 = \alpha * h_2$

...

bis h_i = 0/1 falls h_i = 0 setze h_i auf 1

Bsp: n = 100

h_1 = 0,45454 * 100 = 45,454 = 45

h_2 = 0,45454 * 45 = 20

h_3 = 0,45454 * 20 = 9

h_4 = 0,45454 * 9 = 4

h_5 = 0,45454 * 4 = 1

Inkrementfolge (45, 20, 9, 4, 1)

Def. Shellsort:

Sei $h_1 > h_2 > ... > h_t$ eine streng monoton abnehmende Folge (Inkrementfolge) natürlicher Zahlen mit $h_t = 1$.

Man fühe für a[1], a[2], ... , a[n] nacheinander die h_i−Sortierung nach Insertionsort durch.

Dann heißt dieses Sortierverfahren Shellsort.

Shellsort ist nicht stabil !

Begr: Bei den einzelnen h−Sortierungen kann es vorkommen, daß ein Satz mit gleichem Schlüssel wie ein Satz links von ihm, an ihm vorbeirückt.

Bubblesort

Zwei benachbarte Elemente werden Vertauscht, wenn der Linke echt größer als der recht ist.

=> Bubbelsort ist stabil

p = (6, 1, 2, 5, 4, 3)

	6	1	2	5	4	3	r = 6 = n
1. Dl.	1	6					
		2	6				
			5	6			Durchlauf bricht ab
				4	6		
					3	***6***	Vertauscht r = 5
	1	2	5	4	3	6	
2. Dl.			4	5			
				3	***5***		Vertauscht r = 4
	1	2	4	3	5	6	
3. Dl.			***3***	***4***			Vertauscht r = 3
	1	2	3	4	5	6	
4. Dl.	1	2	3	4	5	6	keine Vertauschung

4 Durchläufe insgesamt

Quicksort

Pivotelement =int((links+rechts)/2); Beim Aufruf *(1, 5) links=1 und rechts=5 ist das Pivotelement 3; i wandert solange von links nach rechts bis es auf ein Element trifft, welches größer/gleich dem Pivotelement ist. j wandert solange von rechts nach links bis es auf ein Element trifft, welches kleiner/gleich dem Pivotelement ist. Dann wird i und j vertauscht und sie wandern eins weiter. Das ganze geht so lange bis i größer j ist. Danach folgt ein neuer Aufruf mit *(left, j) und dann einer mit *(i, right).

p = (1, 5, 4, 2, 3) n = 5

	1	2	3	4	5	
			pivot			
	1	5	4	2	3	Aufruf *(1, 5)
		i			j	
	1	3	4	2	5	i und j vertauscht
			i	j		
	1	3	2	4	5	i und j vertauscht
			j	i		i größer j
	1	3	2			neuer Aufruf *(1, 3)
		i	j			
	1	2	3			i und j vertauscht
		j	i			i größer j
	1	2				neuer Aufruf *(1, 2)
	ij					
	1	2				i und j vertauscht
j		i				i größer j
	1					neuer Aufruf *(1, 0)
		2				neuer Aufruf *(2, 2)
			3			neuer Aufruf *(3, 3)
				4	5	neuer Aufruf *(4, 5)
				ij		
				4	5	i und j vertauscht
			j		i	i größer j
				4		neuer Aufruf *(4, 3)
					5	neuer Aufruf *(5, 5)
	1	2	3	4	5	

Quicksort ist nicht stabil!

Selectionsort

Finde das kleinste Element im Feld und tausche es gegen das an der ersten Stelle befindliche Element aus, finde danach das zweitkleinste Element und tausche es gegen das an der zweiten Stelle befindliche Element aus und fahre in dieser Weise fort, bis das gesamte Feld sortiert ist.

p = (3, 4, 2, 1)

3	4	2	1
1	4	2	3
1	2	4	3
1	2	3	4
1	2	3	4

Selectionsort ist nicht stabil!

Begr.: Ein Element kann durch Vertauschung hinter ein gleichwertiges gelangen. Beim Suchen des Elementes wird jedoch immer zuerst das vordere genommen .

Heapsort

p = (12, 19, 20, 9, 22, 23, 3) n = 7

i =	1.	2.	3.	4.	5.	6.	7.
	12	19	20	9	22	23	3

1. mache p zum Heap

m = int(n / 2) +1 =int(7 / 2) + 1 = 4 alle Element von m bis n haben keine Nachfolger => ist Heap

12	19	20	9	22	23	3	− ist Heap
12	19	20	9	22	23	3	− *sift(3, 7) soll Heap werden
		23	9	22	20	3	♦a[i] > a[2i] && a[i] > a[2i + 1] falls nicht vertauschen mit dem Größeren von beiden (bei Vertau− schung teste ob Bedingung für Nach− folger erfüllt ist, sonst vertausche)
12	19	23	9	22	20	3	− ist Heap / soll Heap werden
12	22	23	9	19	20	3	− *sift(2, 7)
12	22	23	9	19	20	3	− ist Heap / soll Heap werden
23	22	12	9	19	20	3	− *sift(1, 7)
23	22	20	9	19	12	3	
23	22	20	9	19	12	3	ist Heap

2. Sortierte Folge erstellen

Wurzel ist größtes Element

23	22	20	9	19	12	3	
3	22	20	9	19	12 |	23	Vertausche (1, 7)
22	3	20	9	19	12		*sift(1, 6)
22	19	20	9	3	12		
12	19	20	9	3 |	22		Vertausche (1, 6)
20	19	12	9	3			*sift(1, 5)
3	19	12	9 |	20			Vertausche (1, 5)
19	3	12	9				*sift(1, 4)
19	9	12	3				
3	9	12 |	19				Vertausche (1, 4)
12	9	3					*sift(1, 3)
3	9 |	12					Vertausche (1, 3)
9	3						*sift(1, 2)
3 |	9						Vertausche (1, 2)
3	9	12	19	20	22	23	sortierte Folge

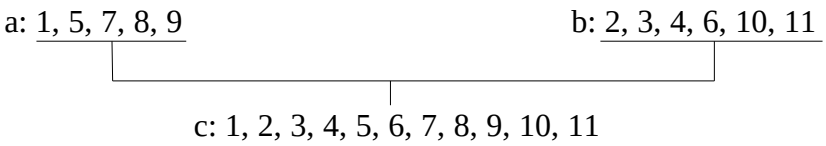
Heapsort ist nicht stabiel!

Anzahl der sift−Aufrufe für n Elemente: m = int(n/2)+1

Anzahl = m + n −3

Sortieren von Tabellen durch Mischen

2−Weg−Mischen



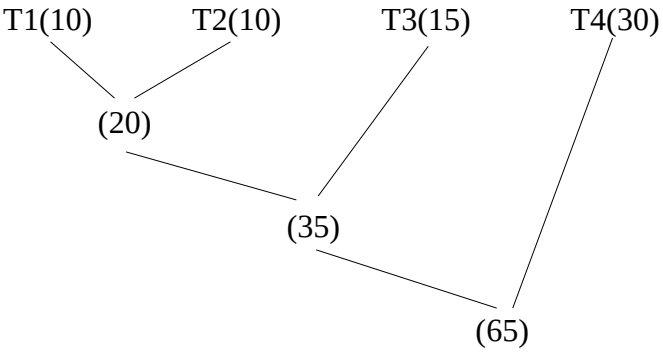
optimale Mischreihenfolge

Greedy − Algorithmus

- bestimme jeweils die beiden kürzesten Tabellen T1 u. T2
- 2 − Weg − Mischen

(solange bis nurmehr eine Tabelle übrig ist)

Bsp.:



Mergesort

Sortieren von einer Tabelle durch wiederholtes 2−Weg−Mischen.

p = (60, 13, 4, 19, 5, 24) n = 6

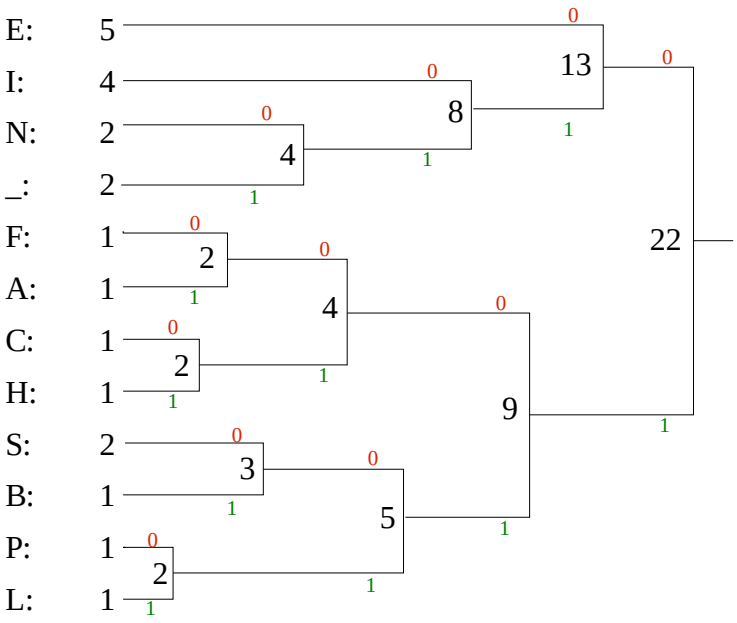
ms = mergeSort

```
60  13  4   19  5   24  *ms (1, 6)
60  13  4               |   *ms (1, 3)
60  13                  |   |   *ms (1, 2)
60                      |   |   |   *ms (1, 1)
                        |   |   |   +
    13                  |   |   |   *ms (2, 2)
                        |   |   |   +
13  60                  |   |   |   merge
                        |   |   +
        4               |   |   *ms (3, 3)
                        |   |   +
4   13  60              |   |   merge
                        |   +
                        |   *ms (4, 6)
            19  5   24  |   |   *ms (4, 5)
            19  5       |   |   |   *ms (4, 4)
            19          |   |   |   +
                5       |   |   |   *ms (5, 5)
                        |   |   |   +
            5   19      |   |   |   merge
                        |   |   +
                        |   |   *ms (6, 6)
                        |   |   +
            5   19  24  |   |   merge
                        |   +
4   5   13  19  24  60  |   merge
                        +
```

Huffman − Codierung

1. Durch Zählen die Häufigkeit jedes Zeichens innerhalb der zu kodierenden Zeichenfolge ermitteln.
2. Aufbau des Kodierung−Tries entsprechend den Häufigkeiten
   - für jeden Wert wird ein Knoten erzeugt
   - immer die beiden kleinsten Knoten werden zu einem Neuen zusammengefasst (bis nurmehr ein Knoten übrig ist)

„EIN EINFACHES BEISPIEL“



	Codetabelle	Bits
E:	00	10
I:	010	12
N:	0110	8
_:	0111	8
F:	1000	4
A:	1001	4
C:	1010	4
H:	1011	4
S:	1100	8
B:	1101	4
P:	1110	4
L:	1111	4
		74

EIN EINFACHES BEISPIEL

00**010**0110**0111**00**010**0110**1000**1001**1010**1011**00**110**0111**1101**00**0101**1100**1110**010**001**1111**

Sortieren von Dateien (externes Sortieren)

Natürliches Mischen

(3−Band−2−Phasen−mischen)

p = (21, 12, 32, 15, 25, 18, 19, 36) n = 7

C:	21	12 32	15 25	18 36			− runs suchen
A:	21	15 25					− runs abwechselnd auf A und B
B:	12 32	18 36					verteilen und neue runs suchen
C:	12 21 32	15 18 25 36					− immer je zwei runs durch 2−Weg−mischen sortieren und neu in C anordnen / von Vorne ...runs suchen
A:	12 21 32						
B:	15 18 25 36						
C:	12 15 18 21 25 32 36						− nur mehr ein run in C => Sortiert

Sequentielle Suche

(siehe Skript S. 62)

Binäres Suchen

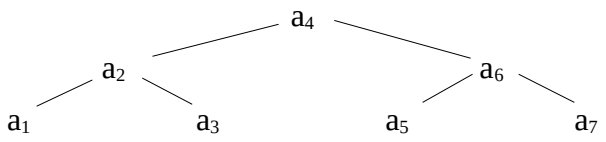
p = (12, 15, 18, 21, 25, 32, 36) n = 7 gesuchter Schlüssel r = 16

1.	2.	3.	4.	5.	6.	7.
12	15	18	21	25	32	36

left	right	m	a[m]	
1	7	4	21	21 > 16 => right = m − 1
1	3	2	15	15 < 16 => left = m + 1
3	3	3	18	18 > 16 => right = m − 1
3	2		left > right (Abbruchbedingung)=> Schlüssel ist nicht enthalten	

m = int((left + right) / 2)

Suchwege als binärer Baum



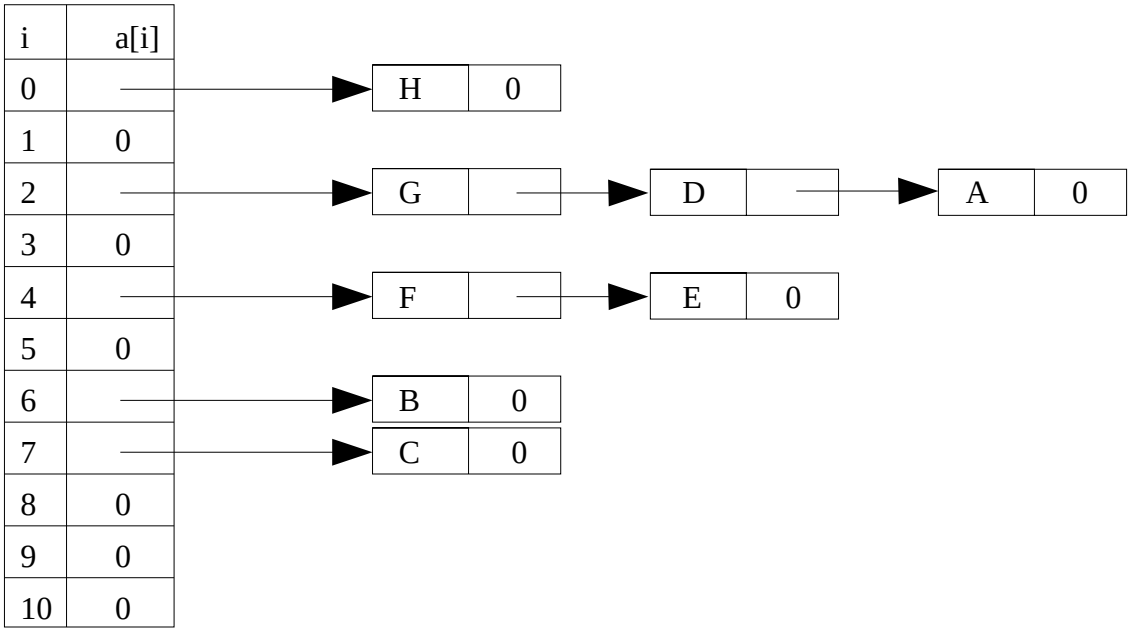
Hash−Strukturen

Separate chaining

m = 11

h in Tabellenform

key	A	B	C	D	E	F	G	H
h(key)	2	6	7	2	4	4	2	0



Coalesced hashing

m = 11

h in Tabellenform

key	A	B	C	D	E	F	G	H	I
h(key)	2	0	3	0	4	10	1	2	0

Falls ein Tabellenplatz belegt ist, wird ein Link auf den nächsten freien Platz von Unten gesetzt.

Aufnahmereihenfolge: A, B, C, D, E, F, G, H, I

	key	link	
0	B	10	
1	G		
2	A	8	
3	C		
4	E		
5			1/m
6			10/m
7	I		
8	H		
9	F	7	
10	D	9	

<u>Aufwand Coalesced Hashing:</u>

m Tabellenlängen

n belegte Plätze

α n/m Belegungsfaktor

− Mittlere Anzahl überprüfter Plätze bei erfolgreicher Suche

$$C_n \approx 1+\frac{e^{2\alpha}-1-2\alpha}{8\alpha}+\frac{\alpha}{4}$$

− Mittlere Anzahl überprüfter Plätze bei erfolgloser Suche oder Einfügen des (n+1)ten Elementes

$$C'_n \approx 1+\frac{e^{2\alpha}-1-2\alpha}{4}$$

Wahrscheinlichkeit, daß freie Plätze als nächstes belegt weden

Linear probing

m = 11

h in Tabellenform

key	A	B	C	D	E	F	G	H
h(key)	2	0	3	0	4	10	1	2

Falls ein Tabellenplatz belegt ist, wird der nächste frei Platz unterhalb genommen (bei Tabellenende wird wieder oben angefangen). Die Suche nach einem Element geht so lange, bis das Element oder ein freier Tabellenplatz gefunden wird oder das Ausgangsfeld wieder erreicht ist.

Aufnahmereihenfolge: A, B, C, D, E, F, G, H

	key	
0	B	
1	D	
2	A	
3	C	
4	E	
5	G	
6	H	
7		9/m
8		1/m
9		1/m
10	F	

Wahrscheinlichkeit, daß freie Plätze als nächstes belegt weden

<u>Aufwand Linear probing:</u>

m Tabellenlängen

n belegte Plätze

α n/m Belegungsfaktor

− Mittlere Anzahl überprüfter Plätze bei erfolgreicher Suche

$$C_n \approx \left(1+\frac{1}{1-\alpha}\right) / \ 2$$

− Mittlere Anzahl überprüfter Plätze bei erfolgloser Suche oder Einfügen des (n+1)ten Elementes

$$C'_n \approx \left(1+\frac{1}{(1-\alpha)^2}\right) / \ 2$$

Double Hashing
Funktioniert genauso wie Linear probing, nur daß zusätzlich eine zweite Hashfunktion (Schrittweitenfunktion) für jedes Element eine Schrittweite errechnet.

m = 13 h(k) = k mod (m) w(k) = (k mod (m−1))+1

Hashfunktion Schrittweitenfunktion

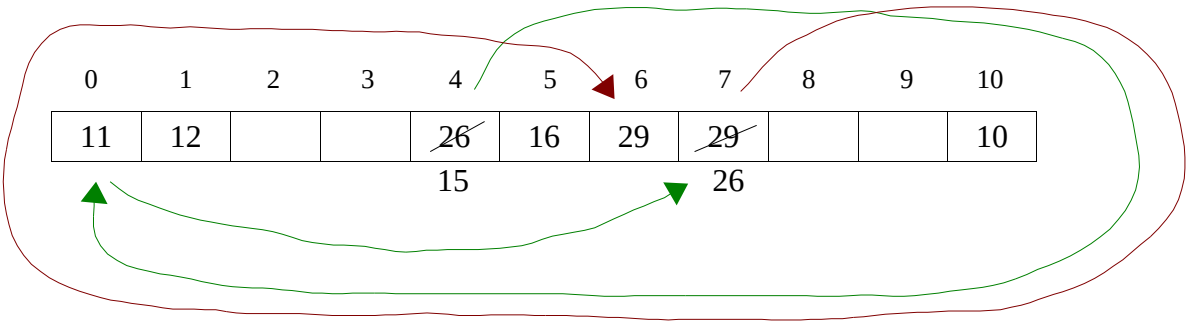
k	84	97	110	111	124	125	126	127	= Auf. reihenfolge
h(k)	6	6	6	7	7	8	9	10	
w(k)	1	2	3	4	5	6	7	8	

		überprüfte Komponenten
0		
1	125	2
2		
3	126	2
4		
5		
6	84	1
7	111	1
8	97	2
9	110	2
10	127	1
11		
12	124	2
		13

Ordered hashing
Ähnlich wie Double hashing, nur daß kollidierende Sätze (aufsteigend) Sortiert eingefügt werden. Immer der Größere wird verschoben. => Suchen geht schneller auf Kosten von Einfügen.

m = 11 h(k) = k mod (m) w(k) = (k mod (m−1))+1

k	10	26	11	29	12	16	15	= Aufnahmereihenfolge
h(k)	10	4	0	7	1	5	4	
w(k)	1	7	2	10	3	7	6	



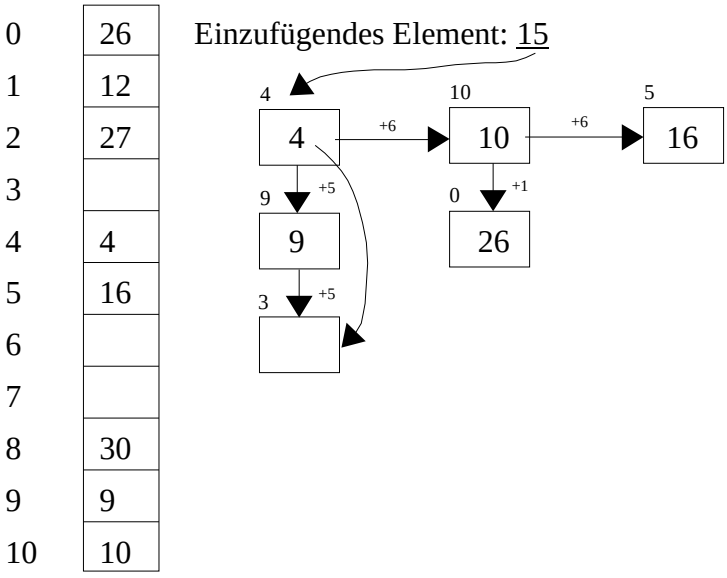
Suche endet, wenn größerer Schlüssel, leerer Platz oder Ausgangsplatz erreicht.

Methode von Brent

Falls Feldplatz belegt ist, wird erst mit der Schrittweite des einzufügenden Elements gesucht ob nächster Platz frei ist. Falls nicht, wird versucht das Element an der ursprünglich einzufügenden Stelle mit seiner Schrittweite zu verdrängen. Geht dies nicht, wird versucht das Element an der Stelle wohin das einzufügende Element mit seiner Schrittweite hinverschoben werden sollte zu verschieben. usw.

m = 11 h(k) = k mod 11 w(k) = (k mod 10)+1

k	10	26	30	12	9	16	27	4	15
h(k)	10	4	8	1	9	5	5	4	4
w(k)	1	7	1	3	10	7	8	5	6



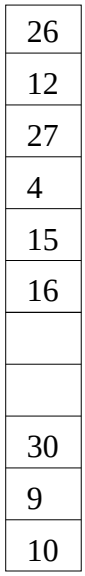


Tabelle nach dem Einfügen der 15

Reihenfolge ob Verdrängung möglich: c = Schrittweite des einzufügenden Elements k