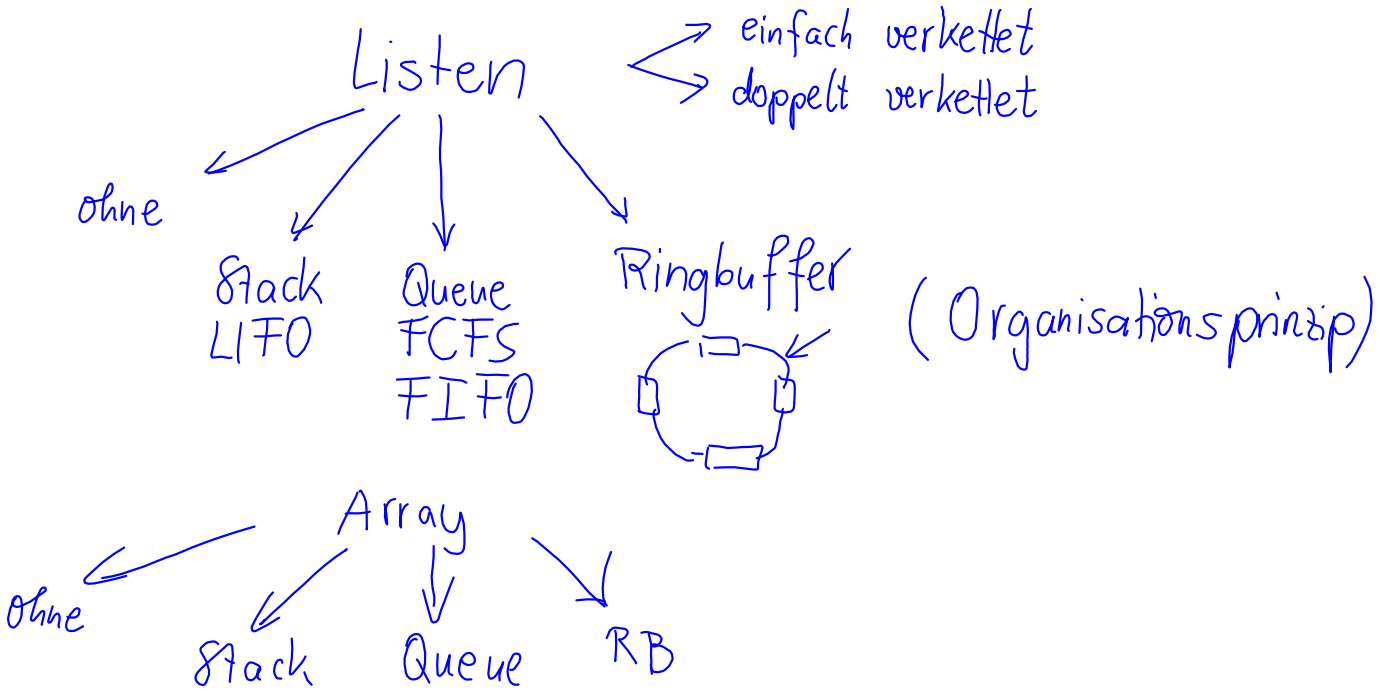
ADT (Abstrakte Datentypen)

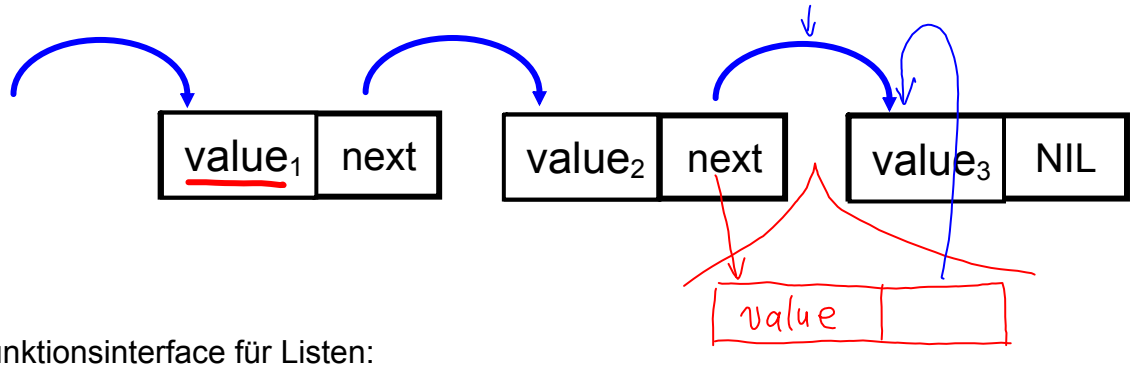
- Listen
- Bäume
- Stack
- Queue
- Ringbuffer
- Hash
- Set / Menge



Listen

Listendefinition (vorwärtsverkettet) :

```
typedef struct item
            { float value;
              struct item *next;
            } itemtyp;

itemtyp *headptr, *aktptr, *tailptr;
```



Funktionsinterface für Listen:

create(); // Liste erstellen

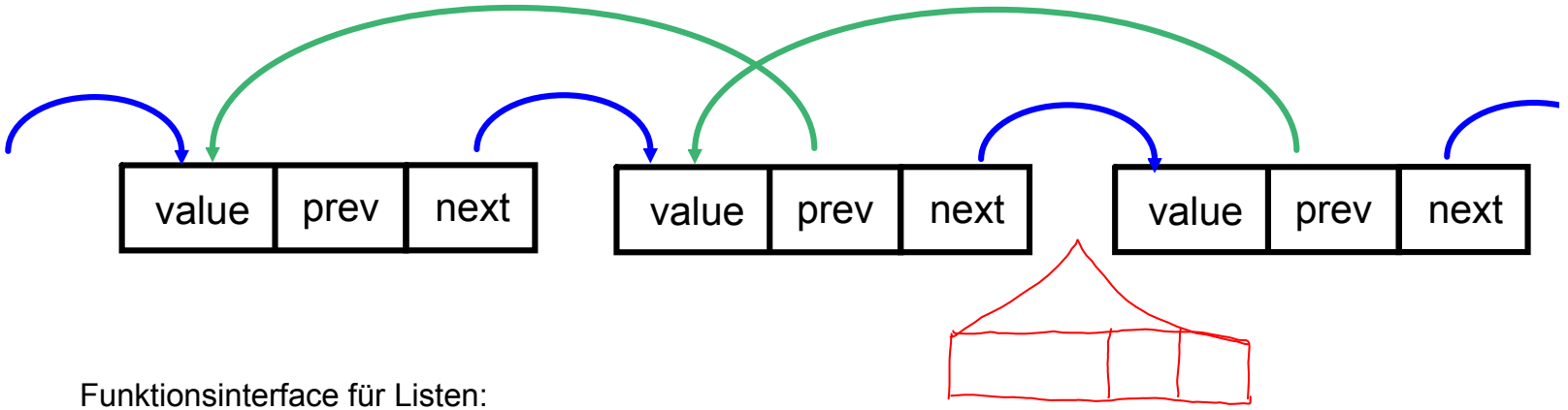
insert(float wert); // Einfügen von wert in die Liste je nach Organisationsprinzip (LIFO, FIFO/FCFS)

search(float wert) // Suche in der Liste nach wert (erfolgreich / erfolglos)

delete(float wert) // Lösche wert (oder je nach Organisationsprinzip erstes/letztes Element)

Listendefinition (doppelt verkettet) :

```
typedef struct item
            { float value;
              struct item *next;
              struct item *prev;
            } itemtyp;

itemtyp *headptr, *aktptr, *tailptr;
```



Funktionsinterface für Listen:

create(); // Liste erstellen

insert(float wert); // Einfügen von wert in die Liste je nach Organisationsprinzip (LIFO, FIFO/FCFS)

search(float wert) // Suche in der Liste nach wert (erfolgreich / erfolglos)

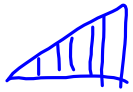
delete(float wert) // Lösche wert (oder je nach Organisationsprinzip erstes/letztes Element)

Sortierung

http://www.sortieralgorithmen.de/index.html
http://programmierung.saschaseidel.de/
https://de.wikipedia.org/wiki/Sortierverfahren
https://de.wikipedia.org/wiki/Selectionsort
https://de.wikipedia.org/wiki/Insertionsort
https://de.wikipedia.org/wiki/Quicksort
https://de.wikipedia.org/wiki/Mergesort
https://de.wikipedia.org/wiki/Bubblesort
https://de.wikipedia.org/wiki/Shakersort
https://de.wikipedia.org/wiki/Shellsort
https://de.wikipedia.org/wiki/Radixsort

geg: $a = (a_1, a_2, \ldots, a_n)$ mit
a_i ordnungsfähig
$a_i < a_j$? muß entscheidbar sein

ges: o.B.d.A sortierte Reihenfolge aufsteigend



$a' = (a_{i_1}, a_{i_2}, \ldots, a_{i_n})$
mit $a_{i_\ell} \leq a_{i_k}$ $\ell < k$

Es gibt n! Reihenfolgen! $O(n!) = O(2^n)$

(Siehe auch systolische Algorithmen)

Def:

interne Sortierverfahren: gesamte Folge verbleibt im Hauptspeicher im wahlfreien Zugr

externe Sortierverfahren: nur ein Teil der Folge verbleibt im Hauptspeicher, der Rest verbleibt auf externen Speichermedien im sequentiellen (langsamen) Zugriff

stabiles Sortierverfahren: Ein Sortierverfahren heißt stabil, wenn gleiche Elementwerte nach dem Sortieren in der gleichen Reihenfolge zueinander bleiben.

bzw. Ein stabiler Sortieralgorithmus ist ein Algorithmus, bei dem die relative Ordnung von Ausgangselementen mit gleichem Wert in der sortierten Folge erhalten bleibt.

indirekte/logische Sortierung: die Elemente des Vektors a verbleiben an ihrer Position lediglich ein Indexvektor IND=(1,2,...n) wird entsprecher des Wertes von a[i] umsortiert.

IND' =(i[1],i[2],...,i[n])

Bsp.: a = (d,a,e,c,a,b,e) ==> a´ = (a,a,b,c,d,e,e)

IND = (1,2,3,4,5,6,7) ==> IND´ =

stabiler Sortieralgorithmus: IND' = (2,5,6,4,1,3,7)

invers stabiler Sortieralgorithmus: IND' = (5,2,6,4,1,7,3)

instabiler Sortieralgorithmus: IND' = (2,5, ,7,3)

IND' = (5,2, ,3,7)

Sortieralgorithmen (Übersicht):

1) Selection Sort (Sortieren durch Auswählen): von jeder Position, das kleinste Element im Rest suchen und nach "vorn" tauschen

2) Insert Sort (Sortieren durch Einfügen): Sortierung durch direktes Einfügen eines Elementes in eine beim Rückwärtsgehen geschaffene passende Lücke

3) Bubble Sort: durch Vergleich und evtl. Vertauschen benachbarter Elemente steigen die großen Elemente wie Blasen ans Ende der Zahlenfolge auf (Variante Shaker-Sort für alternative Richtungen)

4) Shell Sort: wie mittels Bubble Sort werden Elemente miteinander vertauscht. Hierzu wählt man eine gegen 1 fallende Distanzfolge, d.h. abschließend wird Bubble Sort ausgeführt

5) Quick Sort: nach dem Prinzip "Teile und Herrsche" / "Devide and conquer" rekursiv die Datenfolge bzgl. des Medians partitionieren und bis auf einelementige Mengen verzweigen, d.h. in Teilmengen zerlegen und diese separat nach dem gleichen Prinzip sortieren

6) Merge Sort: verteilen auf Bänder und Zusammenmischen in einem modifizierten Reißverschlußverfahren (evtl. Mehrband-Mischen mit mehreren "Bändern")

7) Heap Sort: baumartige Anordnung der Vektorelemente als Heap und "pflücken" des Root-Elementes mit anschließender Heap-Reorganisation

8) Bucket Sort (z.B. für Strings): Strings werden in (z.B. 26) Buckets (ihrer Anfangsbuchstaben) aufgeteilt und dann jeder Bucket für sich sortiert

9) Counting-Sort: verschiedene Algorithmen zum Zählen der Vorgänger v eines Vektorelementes, d.h. das Element steht in der sortierten Folge auf Platz Nummer v+1

10) Radix-Sort: ausgehend von der Zifferndarstellung wird jede Ziffernstelle von der niederwertigen bis zur höchstwertigen Stelle mittels eines stabilen Sortierverfahrens sortiert

1) Selection Sort

$a_j < MIN$ stabil

$a_j \leq MIN$ invers stabil

SelectionSort - Grobentwurf

```
E: n, a[i] i=1(1)n
FOR (i=1;i<n;i++) DO
  { // Suche kleinstes Element im Rest a[i+1]...a[n]
  }
    IF (i != IMIN) THEN {// tausche a[i] mit a[IMIN]}
   }
A: a[i] i=1(1)n
```

<> ≠

SelectionSort - physische Sortierung

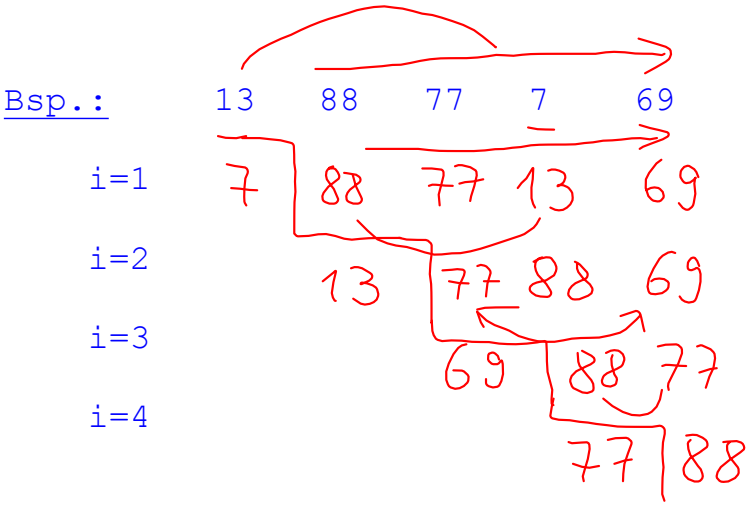
```
E: n, a[i] i=1(1)n
FOR (i=1;i<n;i++) DO
  { MIN=a[i];IMIN=i;
    FOR (j=i+1;j<=n;j++) DO
      { IF (a[j]<MIN) THEN {MIN=a[j];IMIN=j;}
      }
    IF (i != IMIN) THEN
      {a[IMIN]=a[i];a[i]=MIN;}
  }
A: a[i] i=1(1)n
```

SelectionSort - logische Sortierung

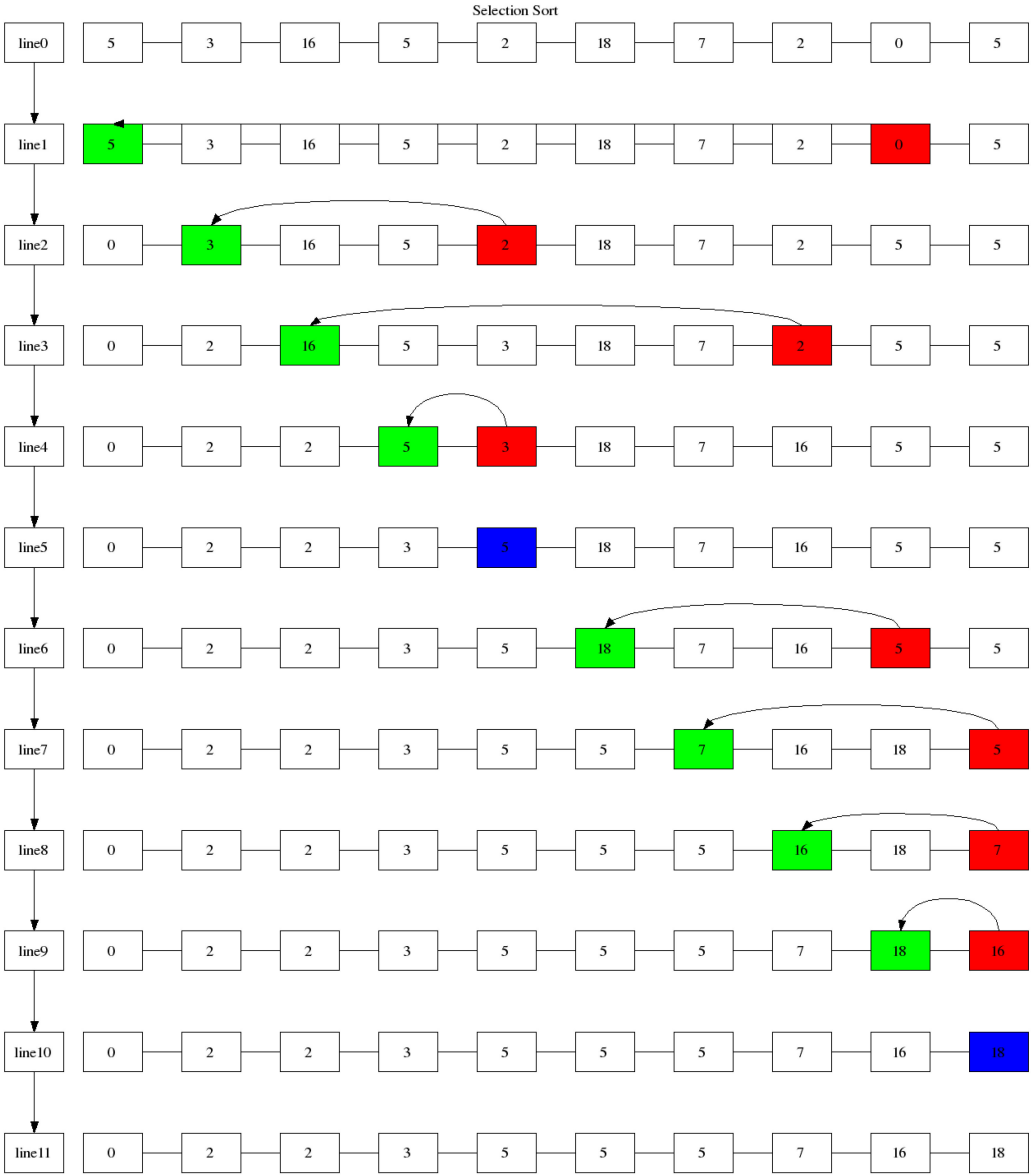
```
E: n, a[i] i=1(1)n
FOR (i=1;i<=n;i++) DO IND[i]=i;
FOR (i=1;i<n;i++) DO
  { MIN=a[IND[i]];IMIN=i;
    FOR (j=i+1;j<=n;j++) DO
      { IF (a[IND[j]]<MIN) THEN {MIN=a[IND[j]];IMIN=j;}
      }
    IF (i != IMIN) THEN
      {tmp=IND[i];IND[i]=IND[IMIN];IND[j]=tmp;}
  }
A: a[IND[i]] i=1(1)n
```

1. 1 2 3

indirekte Adr.


Bsp.:
13 88 77 7 69
i=1
7 88 77 13 69
i=2
13 77 88 69
i=3
69 88 77
i=4
77 88

Selection Sort



2) Insert Sort / Insertion Sort

InsertionSort - Grobentwurf

```
E: n, a[i] i=1(1)n
FOR (i=2;i<=n;i++) DO
  { // Einfügen des i-ten Elementes in die passende beim
    // rückwärtsgehen geschaffene Lücke in a[1]...a[i]
  }
A: a[i] i=1(1)n
```

InsertionSort - physische Sortierung

```
E: n, a[i] i=1(1)n

FOR (i=2;i<=n;i++) DO
  { aa=a[i]; j=i;
    WHILE (a[j]>=aa) && (j>1) DO
      {j--;
       a[j+1]=a[j];
      }
    a[j]=aa;
  }
A: a[i] i=1(1)n
```

InsertionSort - logische Sortierung

```
E: n, a[i] i=1(1)n

FOR (i=1;i<=n;i++) DO IND[i]=i;

FOR (i=2;i<=n;i++) DO

  { aa=a[IND[i]]; j=i;

    WHILE (a[IND[j]]>=aa) && (j>1) DO
      {j--;
       IND[j+1]= IND[j];
      }

    IND[j]=i;  // oder IND[j]=IND[i];
  }

A: a[IND[i]] i=1(1)n
```

Bsp.: 13 88 77 7 69

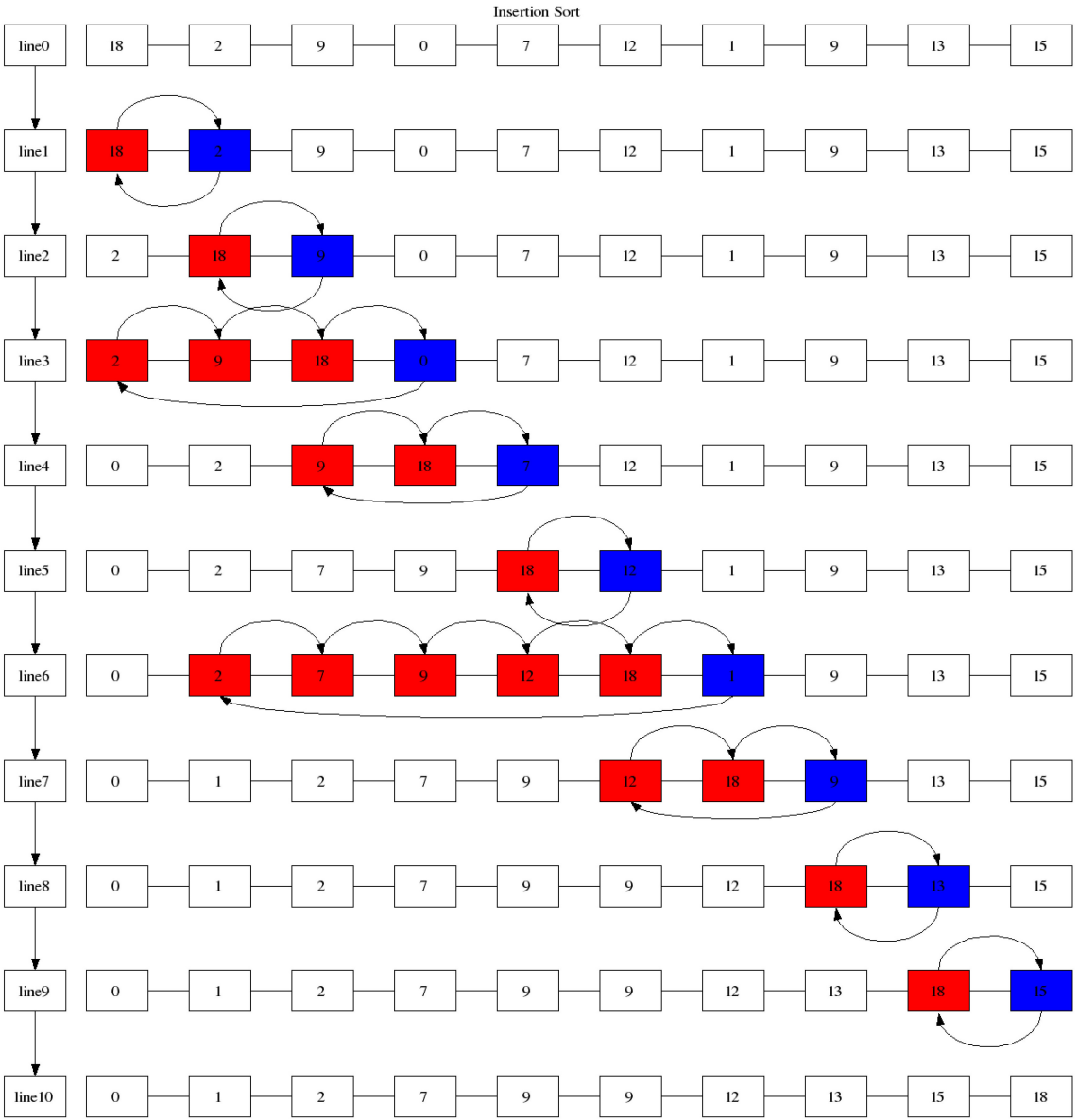
i=2

i=3

i=4

i=5

Insertion Sort



3) Bubble Sort → Shaker Sort

BubbleSort_1 - physische Sortierung

```
E: n, a[i] i=1(1)n
FOR (i=1;i<=n;i++) DO
  { FOR (j=1;j<n-i;j++) DO
      { IF (a[j+1]<a[j]) THEN
         {t=a[j];a[j]=a[j+1];a[j+1]=t;}
      }
   }
A: a[i] i=1(1)n
```

Bsp.: 13 88 77 7 69

i=1

i=2

i=3

i=4

BubbleSort_2 - physische Sortierung

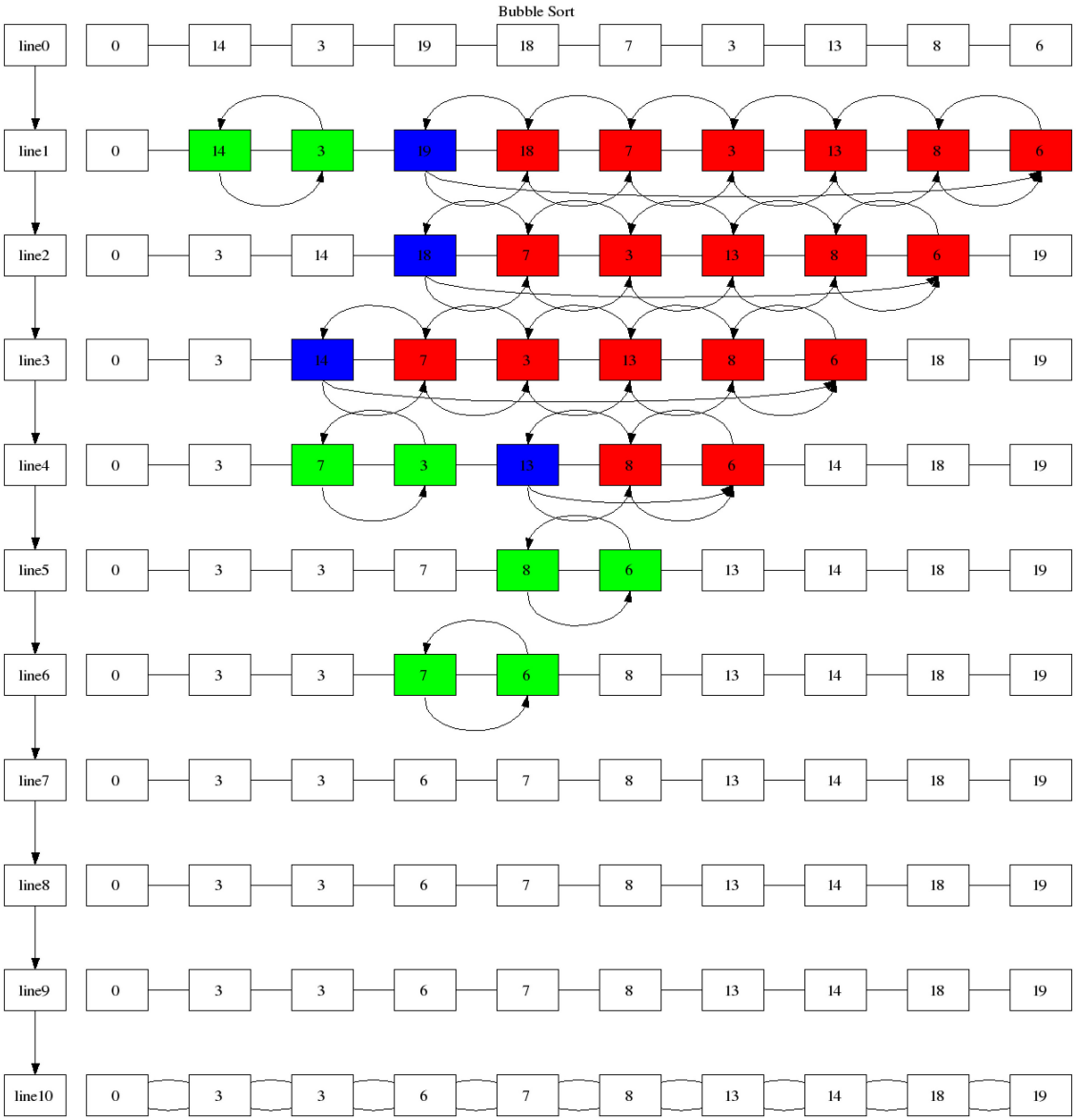
```
E: n, a[i] i=1(1)n
tausch=TRUE;
WHILE tausch DO
  { tausch=FALSE;
    FOR (j=1;j<n;j++) DO
      { IF (a[j+1]<a[j]) THEN
         {t=a[j];a[j]=a[j+1];a[j+1]=t;tausch=TRUE}
      }
   }
A: a[i] i=1(1)n
```

$BC = O(n)$

$WC = O(n^2)$

$AC = O(n^2)$

Bubble Sort


Bubble Sort
line0 0 14 3 19 18 7 3 13 8 6
line1 0 14 3 19 18 7 3 13 8 6
line2 0 3 14 18 7 3 13 8 6 19
line3 0 3 14 7 3 13 8 6 18 19
line4 0 3 7 3 13 8 6 14 18 19
line5 0 3 3 7 8 6 13 14 18 19
line6 0 3 3 7 6 8 13 14 18 19
line7 0 3 3 6 7 8 13 14 18 19
line8 0 3 3 6 7 8 13 14 18 19
line9 0 3 3 6 7 8 13 14 18 19
line10 0 3 3 6 7 8 13 14 18 19

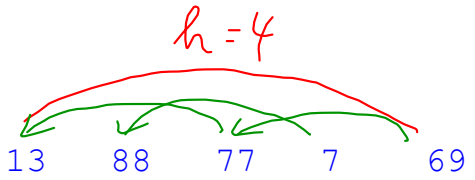
4) Shell Sort

- analog Bubble Sort, aber weiter entfernte Elemente werden miteinander verglichen und evtl. vertauscht
- wähle hierzu eine Folge h[i] von Distanzen, die mit h[1] = 1 endet (d.h. abschließend BubbleSort-Schritte)
- Empfehlung für geeignete Distanzfolgen nach D.E.Knuth
  h[k+1] = 3* h[k] +1 <= n.1 ==> ..., 364, 121, 40, 13, 4, 1
  h[k+1] = 2* h[k] +1 <= n-1 ==> ..., 63, 31, 15, 7, 3, 1
  Startdistanz = größtes h[k] <=n-1

ShellSort_1 - physische Sortierung

```
E: n, a[i] i=1(1)n

h=1; WHILE (3*h+1 < n) DO h=3*h+1;

//alternativ
h=1; WHILE (h < n) DO h=3*h+1;
h = h DIV 3;

WHILE (h>=1) DO
  { FOR (i=h+1;i<=n;i++) DO
      { v=i-h;
        WHILE (v>0) DO
           IF (a[v] > a[v+h]) THEN
              {t=a[i];a[v]=a[v+h];a[v+h]=t;v=v-h;}
      }
    h = h DIV 3;
  }
BubbleSort(); // d.h. h=1


A: a[i] i=1(1)n
```

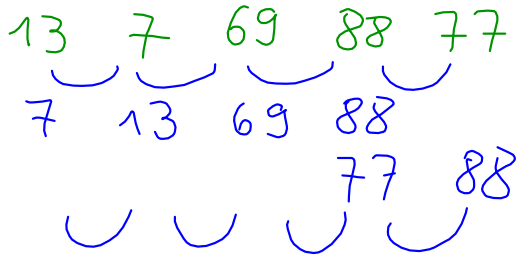
Bsp.:



h=4

h=2

h=1



$O(n^y)$ $1 < y < 2$