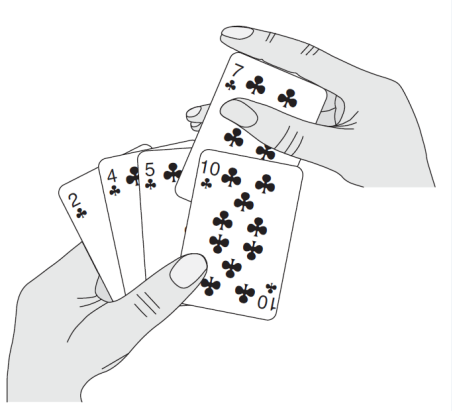
9 Sortieralgorithmen



9.1 Übersicht und Aufgabenstellung

- Sinn und Zweck der Speicherung von Datenbeständen ist es, später auf sie als Ganzes oder auf Teile davon zugreifen zu können
- Sowohl für die Verwaltung eines Datenbestands als Ganzes als auch für den (effizienten) Zugriff auf einzelne seiner Elemente ist es oft zweckmäßig, den Datenbestand nach bestimmten Kriterien zu ordnen, d. h. im Hinblick auf ein bestimmtes Sortierkriterium zu sortieren
- Die Geschichte der Sortierverfahren reicht bis in das 19 Jahrhundert zurück. So hat z. B. Hermann Hollerith für die Auswertung der US-Volkszählung 1860 eine Sortiermaschine für Lochkarten entwickelt, die mittels Fachverteilung arbeitete
- Die ersten, auf Computern auszuführenden Sortieralgorithmen wurden schon 1945 von John von Neumann entwickelt. Er verwendete dabei das von ihm erfundene Mischsortierverfahren (*merge sort*)

Übersicht

- Sortierverfahren, ihre algorithmische Umsetzung und ihre Laufzeitverbesserung sind seit den Anfängen der Informatik Gegenstand der Auseinandersetzung
- Im Laufe der Zeit wurde eine Vielzahl von Verfahren mit sehr unterschiedlichen Effizienzeigenschaften entwickelt
- Minimale Laufzeitkomplexität (Problemkomplexität) beim Sortieren durch Vergleich der Elemente ist $O(n \log n)$
- Die wichtigsten Sortierverfahren sollte man als Softwareentwickler*in kennen!

https://www.youtube.com/watch?v=k4RRi_ntQc8

Einteilung von Sortierverfahren

Interne Sortierverfahren

- Der zu sortierende Datenbestand befindet sich im Arbeitsspeicher
- Die Zugriffszeit ist kurz und von der Lage des Elements unabhängig

Externe Sortierverfahren

- Der zu sortierende Datenbestand befindet sich auf externem Speichermedium (z.B. auf Festplattenspeicher)
- Die Zugriffszeit ist vergleichsweise lang und u. U. von der Lage des Elements abhängig

In diesem Kapitel behandeln wir nur interne Sortierverfahren

Aufgabenstellung

Gegeben ist ein Feld `a[1:n]` von Datenobjekten, die in `a[i].key` ein Schlüsselattribut enthalten

```
type
  KeyType = …
  DataType = …
  ElementType = compound
    key:  KeyType
    data: DataType
  end -- ElementType
```

mind. Operator <

Schlüsselattribut

```
const
  n = … -- n ≥ 1
type
  ArrayType = array [1:n] of ElementType
var
  a: ArrayType
```

Schlüsselattribut (Sortierkriterium/Sortierschlüssel) key ist die Grundlage zur Bildung einer Ordnung

Gesucht ist Permutation, so dass gilt a[1].key ≤ a[2].key ≤ ….. ≤ a[n - 1].key ≤ a[n].key

Aufgabenstellung

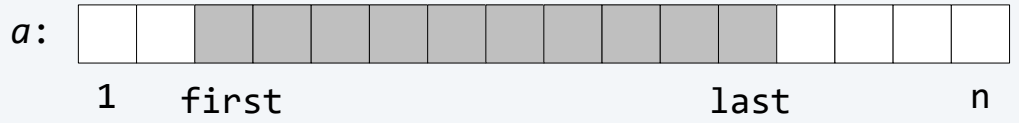
Beispiele für zu sortierende Datenbestände

Feld	**Schlüsselattribut**	**Ordnung**
Telefonbuch	Nachname	lexikographische Ordnung
Klausurergebnisse	Note	≤ auf {1, 2, 3, 4, 5}
Lexikon	Stichwort	lexikographische Ordnung
Studentenverzeichnis	Personenkennzeichen	≤ auf Personenkennzeichen
Fahrplan	Abfahrtszeit	"früher als"

Schnittstelle für Sortieralgorithmen (z.B. auch für teilweise gefüllte Felder)

```
Sort(↓↑a: ArrayType ↓first: int ↓last: int)
```

Problemgröße = `last - first + 1`



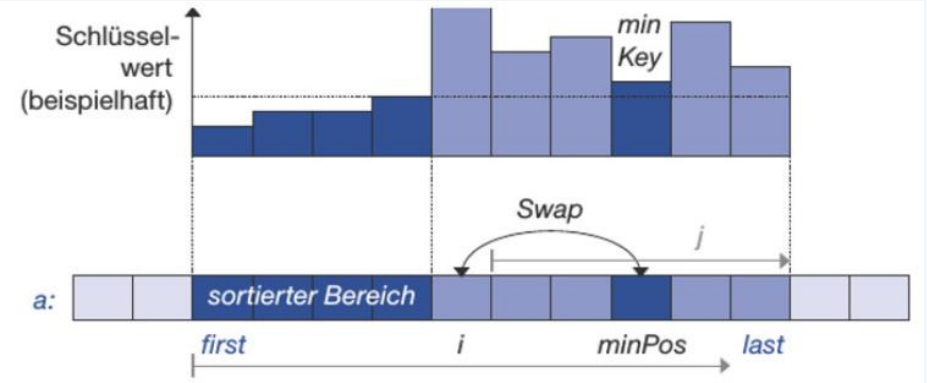
Vorbedingung: `1 ≤ first ≤ last ≤ n`

9.2 Das Auswahlsortierverfahren

Lösungsidee für das Auswahlsortieren (*selection sort*):

- Zu Beginn wird das „kleinste“ Element aus dem zu sortierenden Feld ermittelt und mit dem ersten Element vertauscht (*swap*)
- Danach wird mit dem Rest des Feldes wiederum so verfahren

Wenn man nach dieser Strategie vorgeht, ergibt sich nach einiger Zeit der Zustand:

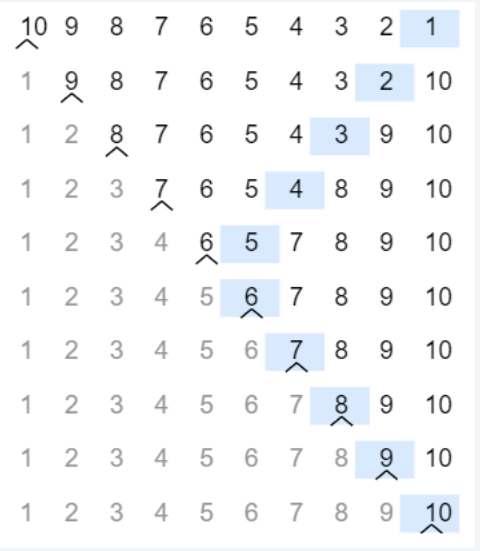


Mit jedem Sortierschritt wächst also der sortierte Bereich und schrumpft der unsortierte Bereich um jeweils ein Element

Ablauf des Auswahlsortierverfahrens

zufällige Folge 1

zufällige Folge 2



absteigend sortierte Folge

Legende:

1 … Einfügeposition `i`

4 … kleinstes Element aus Bereich `a[i]`, …, `a[n]`

Das Auswahlsortierverfahren

Algorithmus: SelectionSort

```
SelectionSort(↓↑a: ArrayType ↓first: int ↓last: int)
  var
    minPos, i, j: int
    minKey: KeyType
begin
  for i := first to last - 1 do
    minPos := i
    minKey := a[minPos].key
    for j := i + 1 to last do
      if a[j].key < minKey then
        minPos := j
        minKey := a[minPos].key
      end -- if
    end -- for
    Swap(↓↑a[i] ↓↑a[minPos])
  end -- for
end SelectionSort
```

Analyse des Laufzeitverhaltens des `SelectionSort`

Algorithmus: `SelectionSort`

```
SelectionSort(↓↑a: ArrayType ↓first: int ↓last: int)
  var
    minPos, i, j: int
    minKey: KeyType
begin
  for i := first to last - 1 do
    minPos := i
    minKey := a[minPos].key
    for j := i + 1 to last do
      if a[j].key < minKey then
        minPos := j
        minKey := a[minPos].key
      end -- if
    end -- for
    Swap(↓↑a[i] ↓↑a[minPos])
  end -- for
end SelectionSort
```

Vergleiche

Vertauschungen

Vergleiche:

$n = last - first + 1$

i	Vergleiche
1	n – 1
2	n – 2
3	n – 3
…	
n – 2	2
n – 1	1

$$\frac{n \cdot (n-1)}{2}$$

Vertauschungen:

$n - 1$

Analyse des Laufzeitverhaltens des `SelectionSort`

- Anzahl der erforderlichen Vergleiche

$$(n-1)+(n-2)+\cdots+2+1=\frac{n\cdot(n-1)}{2}\Rightarrow O(n^2)$$

- Anzahl der Vertauschungen

$$(n-1)\Rightarrow O(n)$$

- Die asymptotische Laufzeitkomplexität ist also $O(n^2)$, sie ist unabhängig von Anordnung der zu sortierenden Datenobjekte, d. h. eine Vorsortierung bringt nichts
- Jedes Datenobjekt wird genau einmal bewegt (günstig bei großen Datenobjekten)
- Optimierung (Vertausch nur bei unterschiedlichen Datenobjekten)

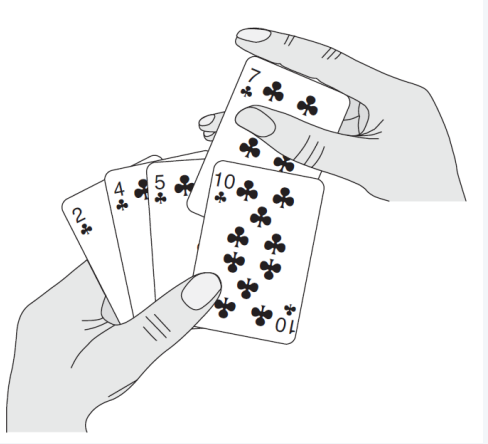
```
if i ≠ minPos then
  Swap(↓↑a[i] ↓↑a[minPos])
end -- if
```

9.3 Das Einfügesortierverfahren

- Das Laufzeitverhalten des `SelectionSort` ($O(n^2)$) ist unbefriedigend
- Überlegung: Man könnte eine ggf. vorliegende Vorsortierung ausnutzen

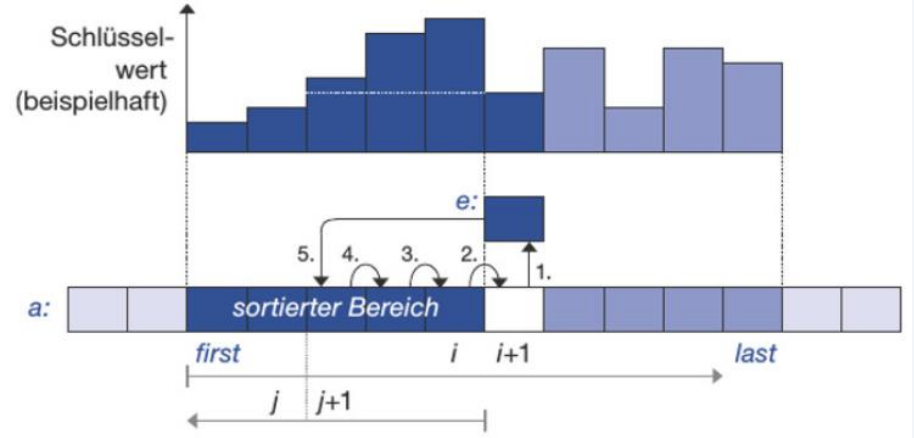
Lösungsidee für das Einfügesortierverfahren (`InsertionSort`):

- Zu Beginn betrachtet man das erste Element des Felds als sortierten Bereich und man sortiert das zweite Element, je nach seinem Schlüsselwert, vor oder hinter dem ersten ein. Der sortierte Bereich wird damit um ein Element vergrößert
- Mit dem nächsten Element verfährt man ebenso: man fügt es im sortierten Bereich an der richtigen Stelle ein

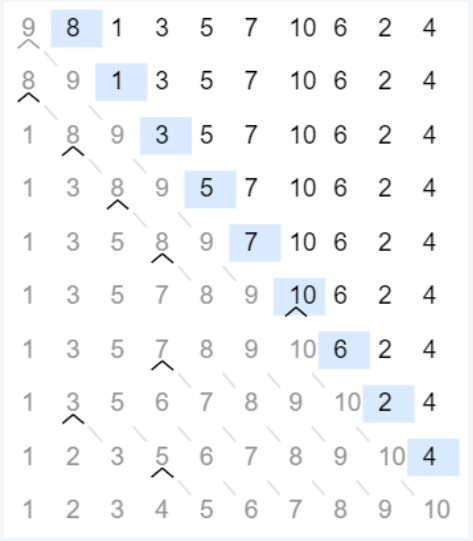


9.3 Das Einfügesortierverfahren

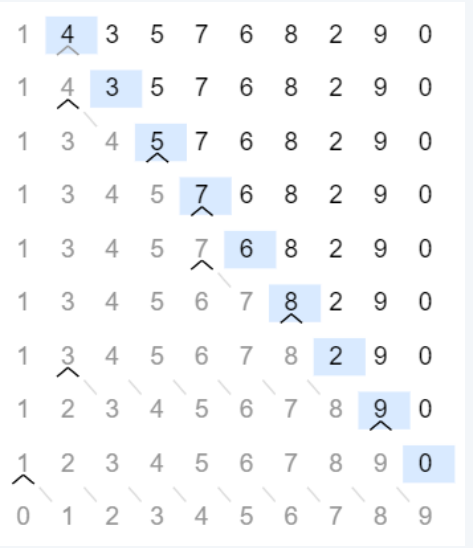
- Wenn man nach dieser Strategie vorgeht, so ergibt sich nach einiger Zeit der Zustand:



Ablauf des Einfügesortierverfahrens



zufällige Folge 1



zufällige Folge 2

absteigend sortierte Folge

Legende:

1 … Einfügeposition i

4 … nächstes Element

Algorithmus für das Einfügesortierverfahren

Algorithmus: `InsertionSort`

```
InsertionSort(↓↑a: ArrayType ↓first: int ↓last: int)
  var
    i, j: int
    e: ElementType
begin
  for i := first to last - 1 do
    -- a[first], ... a[i] is sorted
    e := a[i + 1]
    j := i
    while (j ≥ first) and (e.key < a[j].key) do
      a[j + 1] := a[j]
      j := j - 1
    end -- while
    a[j + 1] := e
    -- a[first], ... a[i + 1] is sorted
  end -- for
end InsertionSort
```

Analyse des Laufzeitverhaltens des `InsertionSort`

Algorithmus: `InsertionSort`

```
InsertionSort(↓↑a: ArrayType ↓first: int ↓last: int)
  var
    i, j: int
    e: ElementType
begin
  for i := first to last - 1 do
    -- a[first], ... a[i] is sorted
    e := a[i + 1]
    j := i
    while (j ≥ first) and (e.key < a[j].key) do
      a[j + 1] := a[j]
      j := j - 1
    end -- while
    a[j + 1] := e
    -- a[first], ... a[i + 1] is sorted
  end -- for
end InsertionSort
```

Vergleiche

Zuweisungen

Vergleiche: $(n-1)+v$

$n = last - first + 1$

$v =$ Durchläufe der inneren Schleife

Zuweisungen:

$2 \cdot (n-1)+v$

Analyse des Laufzeitverhaltens des `InsertionSort`

Vergleiche: $(n-1)+v$ Zuweisungen: $2\cdot(n-1)+v$

Werte für für v:

- sortiert: $v=0$
- absteigend sortiert: $v=\frac{n\cdot(n-1)}{2}$

	sortiert	Durchschnitt	absteigend sortiert
Vergleiche	$n-1$	$\approx n^2/4$	$\approx n^2/2$
Zuweisungen	$2\cdot(n-1)$	$\approx n^2/4$	$\approx n^2/2$

- Für den Durchschnitt wird angenommen, dass ein Element im Schnitt in der Mitte eingefügt wird
- Das Laufzeitverhalten ist im Allgemeinen schlechter als das des Auswahlverfahrens (wegen quadratischer Zahl von Zuweisungen)
- Bei Vorsortierung ist das Laufzeitverhalten aber deutlich besser

Vergleich von `SelectionSort` und `InsertionSort`

Anzahl der Vergleiche

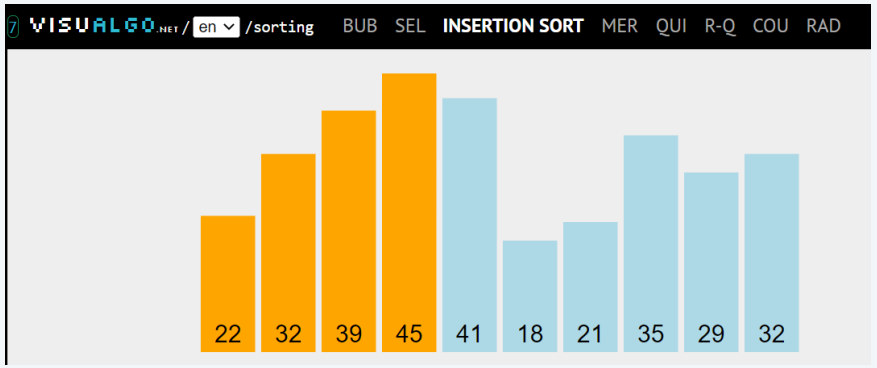
Algorithmus	Günstiger Fall	Durchs. Fall	Ungünstiger Fall
`SelectionSort`	$n^2/2$	$n^2/2$	$n^2/2$
`InsertionSort`	n	$n^2/4$	$n^2/2$

Anzahl der Zuweisungen

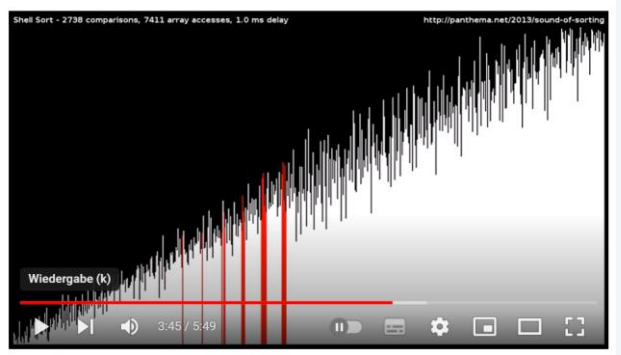
Algorithmus	Günstiger Fall	Durchs. Fall	Ungünstiger Fall
`SelectionSort`	$3 \cdot (n-1)$	$3 \cdot (n-1)$	$3 \cdot (n-1)$
`InsertionSort`	$2 \cdot (n-1)$	$n^2/4$	$n^2/2$

Fazit: Die beiden bisher vorgestellten Sortieralgorithmen sind nur für kleine n geeignet

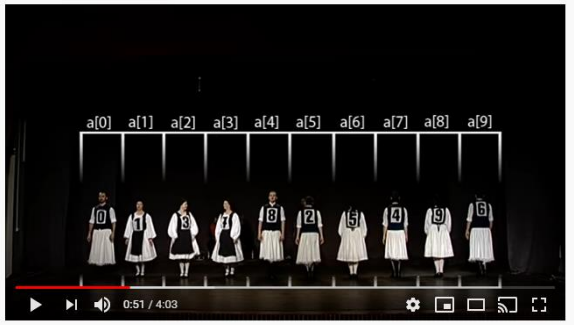
Visualisierungen von Sortieralgorithmen



https://visualgo.net/en/sorting



https://www.youtube.com/watch?v=kPRA0W1kECg



Insert-sort with Romanian folk dance

https://www.youtube.com/watch?v=EdIKIf9mHk0

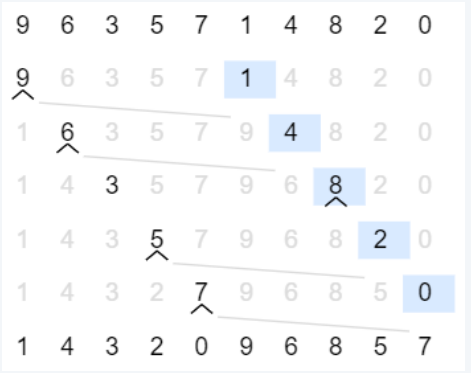
9.4 Das Shell-Sortierverfahren

- Das unbefriedigende Laufzeitverhalten des Einfügesortierens liegt daran, dass für jedes Element, das von seiner Zielposition im sortierten Feld weit entfernt ist, eine entsprechend große Zahl von Elementen jeweils um eine Position nach hinten kopiert werden muss, um vorne Platz für das einzufügende Element zu schaffen.
- Um bei der Anzahl der erforderlichen Zuweisungen eine signifikante Reduktion zu erzielen, müssen die Distanzen, um die Elemente verschoben werden müssen, verringert werden.
- Donald L. Shell hat ein Verfahren dafür vorgeschlagen, das darauf abzielt, diese Nachteile zu verringern. Das Verfahren ist unter der Bezeichnung Shell-Sortieren (ShellSort) bekannt geworden.

Lösungsidee nach Shell

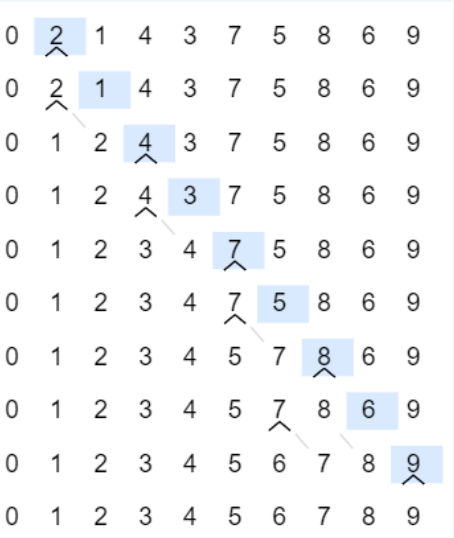
- Die Grobanalyse des Einfügesortierens für den günstigsten Fall, d. h. eines vollständig sortierten Felds hat ergeben, dass man in diesem Fall mit $(n-1)$ Vergleichen und mit $2 \cdot (n-1)$ Zuweisungen auskommt; d.h. dass in diesem Fall das Verfahren von der Ordnung n ist.
- Auf Basis ähnlicher Überlegungen hat Shell seinen Algorithmus so konstruiert, dass dieser in mehreren Schritten eine so gute „Vorsortierung“ des Felds herstellt, dass eine abschließende Sortierung mittels Einfügesortieren mit nur linear ansteigendem Aufwand möglich ist.
- ShellSort (H.D. Shell, 1959): Sortieren von Teilfolgen mit Schrittweite m:
  - Beginne mit $m = n/2$
  - und halbiere m nach jedem Durchlauf
- Sortieren der Teilfolgen erfolgt mittels Einfügesortieren

Beispiel: ShellSort mit n = 10 (unsortierte Folge)



m = 5

m = 2

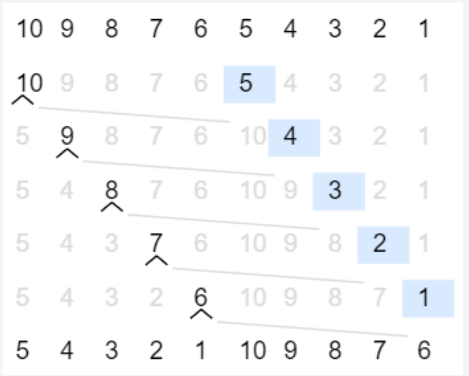


m = 1

Grad der Sortierung wird schrittweise verbessert

Im letzten Durchlauf mit $m = 1$ nur noch wenige Verschiebungen

Beispiel: ShellSort mit n = 10 (absteigend sortierte Folge)



m = 5

m = 2

m = 1

Grad der Sortierung wird schrittweise verbessert

Im letzten Durchlauf mit $m = 1$ keine Verschiebungen!

Algorithmus für das Shell-Sortierverfahren

Algorithmus lässt sich unmittelbar aus dem Einfügesortieren ableiten

beginne mit Schrittweite `m = n div 2` und halbiere Schrittweite bis `m = 1`

```
InsertionSort(↓↑a: array [1:n])



  for i := 1 to n - 1 do
      h := a[i + 1]
      j := i
      while j > 0 and a[j] > h do
         a[j + 1] := a[j]
         j := j - 1
      end -- while
      a[j + 1]:= h
  end -- for
end InsertionSort
```

```
ShellSort(↓↑a: array[1:n])
begin
  m := n div 2
  while m > 0 do
    for i := 1 to n - m do
       h := a[i + m]
       j := i
       while j > 0 and a[j] > h do
         a[j + m] := a[j]
         j := j - m
       end -- while
       a[j + m] := h
    end -- for
    m := m div 2
  end -- while
end ShellSort
```

Algorithmus für das Shell-Sortierverfahren

Algorithmus: `ShellSort`

```
ShellSort(↓↑a: ArrayType ↓first: int ↓last: int)
  var
    i, j, m: int
    e: ElementType
begin
  m := (last – first + 1) div 2
  while m > 0 do
    for i := first to last - m do
      e := a[i + m]
      j := i
      while (j ≥ first) and (e.key < a[j].key) do
        a[j + m] := a[j]
        j := j - m
      end -- while
      a[j + m] := e
    end -- for
    m := m div 2
  end -- while
end ShellSort
```

Analyse des Laufzeitverhaltens des `ShellSort`

Zur Wahl der Schrittweise für die Vorsortierung

- Jede absteigende Folge von m führt zum Ziel
- Die von Shell angegebene Folge $m = {}^{n}/_{2}, {}^{n}/_{4}, {}^{n}/_{8}, \cdots$ ist nicht die beste Wahl
- Bessere Folgen sind (z.B. nach Knuth):

  1, 3, 7, 15, … $m = 2^k - 1$

  1, 4, 13, 40, … $m = \left(3^k - 1\right)/2$

  1, 2, 3, 5, 8, … Fibonacci-Zahlen

Schwierige Berechnungen abhängig von Wahl der Schrittweite m

- Shell gibt an, dass die asympt. Laufzeitkomplexität $O(n^{1.226})$ ist, das ist jedoch zu optimistisch
- Knuth gibt an, dass $O(n^{1.25})$ erreichbar ist
- Sedgewick weist experimentell eine asymt. Laufzeitkomplexität von $O(n^{1.333})$ nach

9.5 Das Quicksort-Verfahren C.A.R. Hoare

Motivation

- Wir wissen, dass die minimale Laufzeitkomplexität von vergleichs-basierten Sortierverfahren $O\big(n \log(n)\big)$ ist, d. h. es muss noch bessere Verfahren als den ShellSort geben
- Eine Lösungsidee dazu stammt von C. A. R. Hoare (a. d. Jahr 1962)

Hoare hat einen Sortieralgorithmus vorgestellt, der zurecht den Namen Quicksort trägt, weil er in den meisten Fällen im Vergleich zu allen anderen Sortierverfahren das günstigste Laufzeitverhalten aufweist.

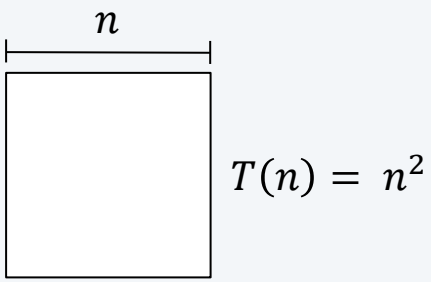
Der Algorithmus von Hoare basiert – wie der binäre Suchalgorithmus – auf der Strategie Teile und Herrsche.

9.5 Das Quicksort-Verfahren C.A.R. Hoare

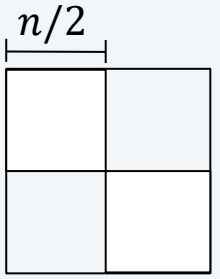
- Das Teilen besteht darin, dass man aus dem zu sortierenden Feld a mit n > 1 Elementen ein beliebiges Element mit dem Schlüsselwert x auswählt und die Feldelemente so umordnet, dass zwei Teilfelder entstehen:
  - Eines, das nur Elemente mit Schlüsselwerten $\leq$ x und ein
  - zweites, das nur Elemente mit Schlüsselwerten $\geq$ x enthält.
- Das Herrschen besteht darin, dieses Verfahren des Aufteilens (rekursiv) auf die beiden Teilfelder anzuwenden (bis keine weitere Teilung mehr möglich ist)

Laufzeitverhalten ohne und mit Teilung

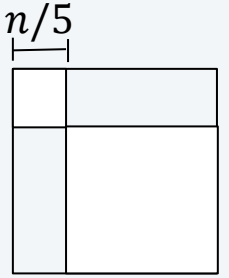
- Größter Gewinn, wenn beide Teile gleich groß sind $n_1 \approx n_2$



$T(n) = n^2$



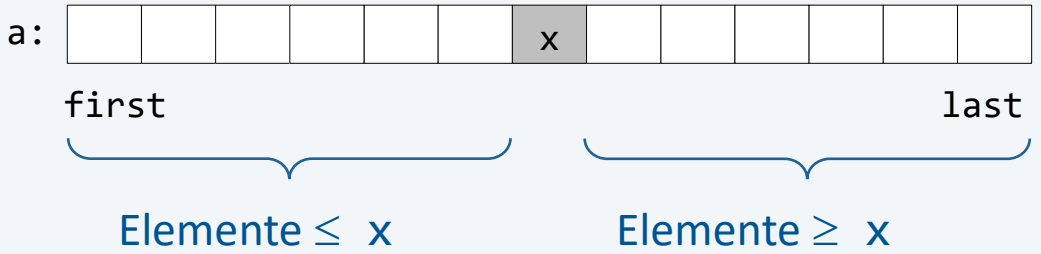
$T(n) = \frac{n^2}{2}$



$T(n) = \left(\frac{n}{5}\right)^2 + \left(\frac{4n}{5}\right)^2 = \frac{17n^2}{25}$

Zur Lösungsidee des Quicksort-Verfahrens

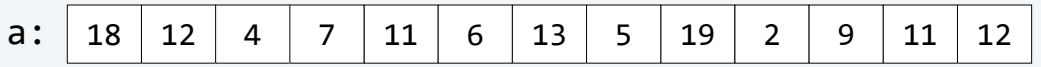
Zerlegung des Felds in zwei Teilfelder, sodass gilt:



Es ist daher eine Umordnung der Feldelemente erforderlich

Beispiel

- Ursprünglicher Inhalt



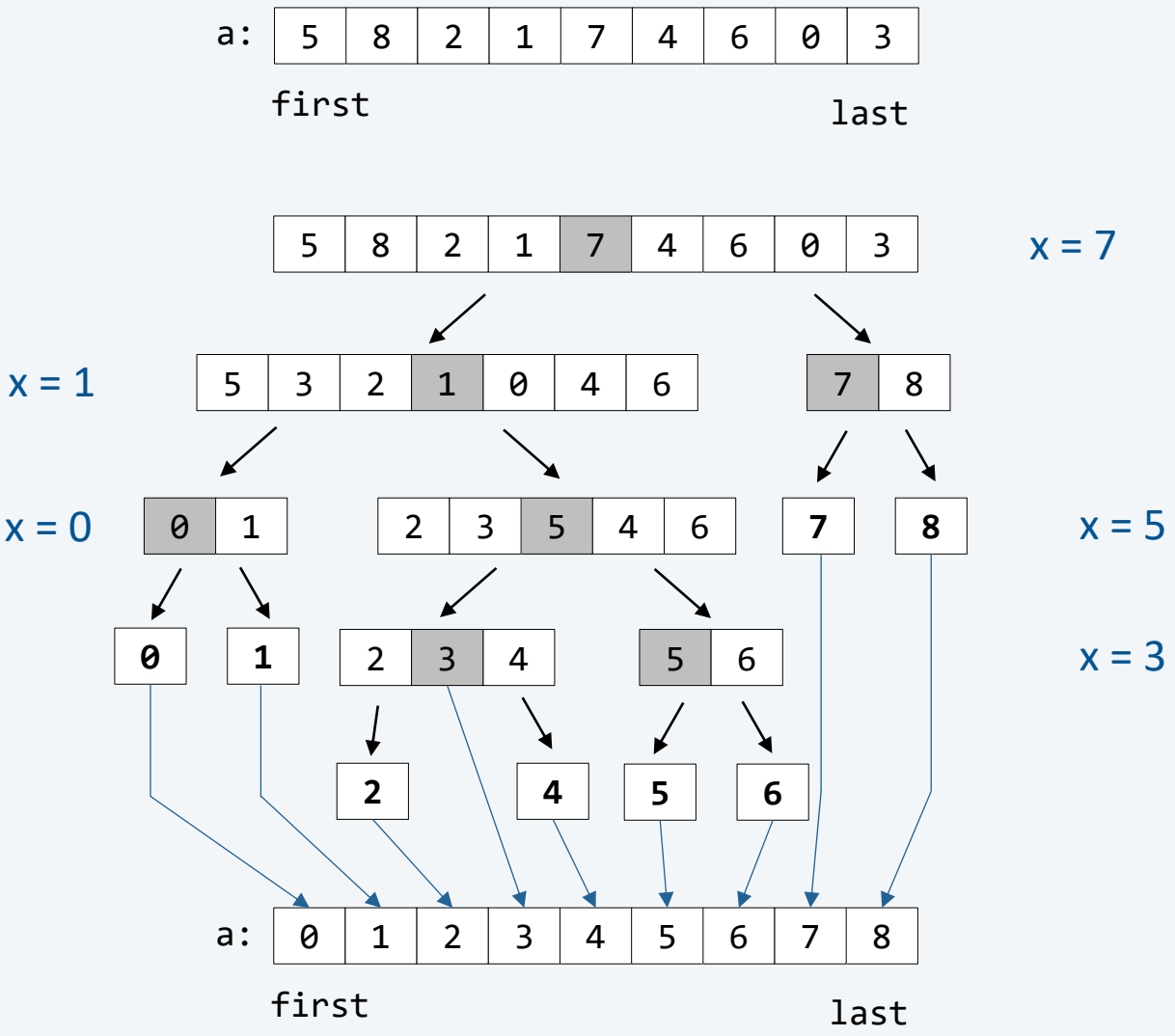
- möglicher Inhalt nach Umordnung (wenn x = 11 ist):

Zur Lösungsidee des Quicksort-Verfahrens

Das Feld `a[first:last]`, mit `first < last`, soll sortiert werden:

- Wähle beliebiges Element mit Schlüsselwert x aus Feld
- Zerlege das Feld `a[first:last]` in zwei Teile, so dass
  - alle Elemente in `a[first:i].key` kleiner gleich als x sind
  - alle Elemente in `a[i+1:last].key` größer gleich x sind
- Wenn das Teilfeld `a[first:i]` noch mehr als c Elemente hat, zerlege das Feld wieder wie oben angegeben, sonst sortiere dieses Teilfeld
- Wenn das Teilfeld `a[i+1:last]` noch mehr als c Elemente hat, zerlege das Feld wieder wie oben angegeben, sonst sortiere dieses Teilfeld

Ablauf des Quicksort-Verfahrens

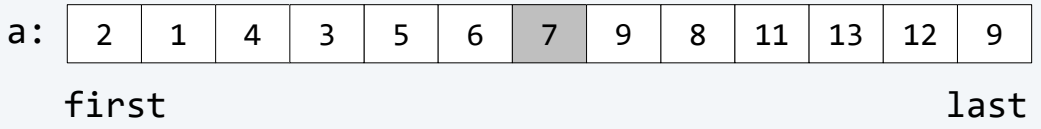


Zur Wahl des Teilungselementes x

Effizienz von `QuickSort` hängt von Wahl des Teilungselements ab

- `x := a[first].key` … schlecht bei vorsortierten Folgen

- Besser: `x := a[(first + last) div 2].key`



Bildung der Teilfelder: Partitionierung (Umordnung)

- Links und rechts von x nach Elementen suchen, die in der „falschen Hälfte“ liegen
  1. suche von links das erste Element `a[i].key` $\geq$ `x`
  2. suche von rechts das erste Element `a[j].key` $\leq$ `x`
  3. vertausche Elemente `a[i]` und `a[j]`
  4. wiederhole (1), (2), (3) mit `i := i + 1` und `j := j - 1` bis `i > j`
- Während der Partitionierung gilt: `first` $\leq$ `i` $\leq$ `j` $\leq$ `last`

  `a[`**`first`**`].key, ..., a[`**`i - 1`**`].key` $\leq$ `x` und

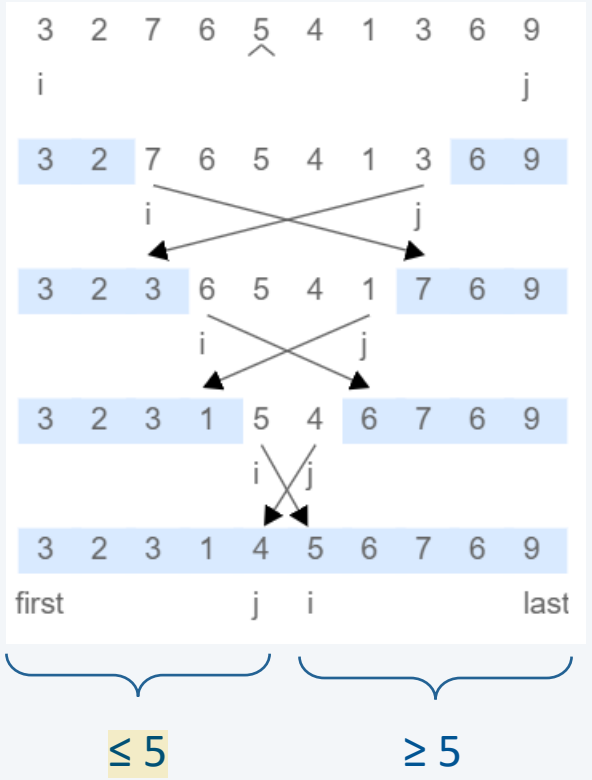
  `a[`**`j + 1`**`].key, ..., a[`**`last`**`].key` $\geq$ `x`
- Nach der Partitionierung gilt: `first` $\leq$ `j` < `i` $\leq$ `last`

  `a[`**`first`**`].key, ..., a[`**`j`**`].key` $\leq$ `x` und

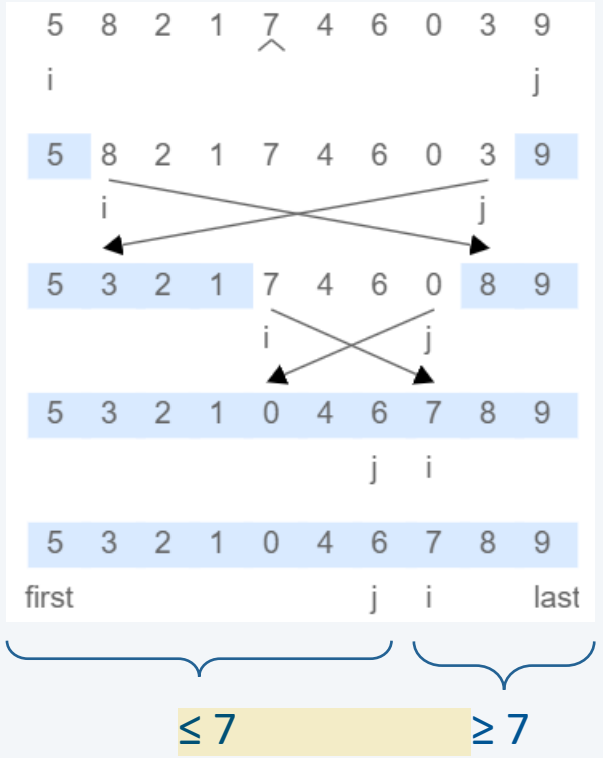
  `a[`**`i`**`].key, ..., a[`**`last`**`].key` $\geq$ `x`

Beispiel zur Lösungsidee für die Partitionierung

x = 5



x = 7



Sortieren der Teilfelder

Nach der Partitionierung gilt: `first ≤ j < i ≤ last`

`a[`**`first`**`].key, ..., a[`**`j`**`].key ≤ x` und

`a[`**`i`**`].key, ..., a[`**`last`**`].key ≥ x`

Rekursive Aufrufe um Teilfelder zu sortieren:

```
QuickSort(↓↑a ↓first ↓j)
QuickSort(↓↑a ↓i ↓last)
```

Aufruf um vollständiges Feld zw. `first` und `last` zu sortieren

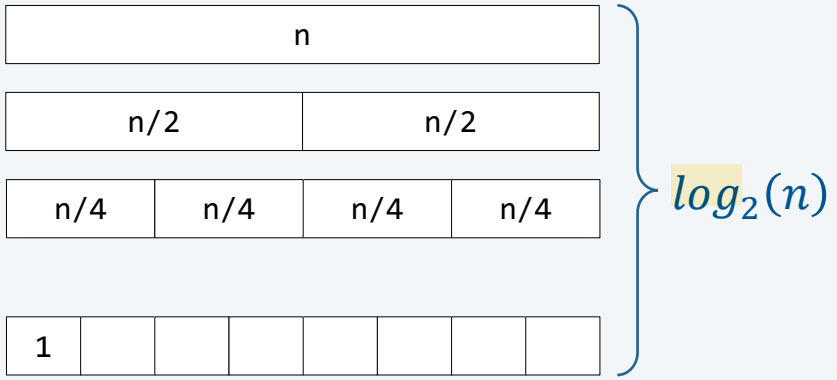
```
var
  a: ArrayType

QuickSort(↓↑a ↓first ↓last)
```

Transformation der Lösungsidee in einen Algorithmus

```
QuickSort(↓↑a: ArrayType ↓first: int ↓last: int)
  var
    x: KeyType
    i, j: int
begin
  if first < last then
    x := a[(first + last) div 2].key
    i := first
    j := last
    repeat
      while a[i].key < x do i := i + 1 end
      while x < a[j].key do j := j - 1 end
      if i ≤ j then
        Swap(↓↑a[i] ↓↑a[j])
        i := i + 1
        j := j - 1
      end -- if
    until i > j
    QuickSort(↓↑a ↓first ↓j   )
    QuickSort(↓↑a ↓i     ↓last)
  end -- if
end QuickSort
```

Analyse des Laufzeitverhaltens

Günstiger Fall:



- Bei jedem Durchlauf n Vergleiche
- Durchläufe: $log_2(n)$

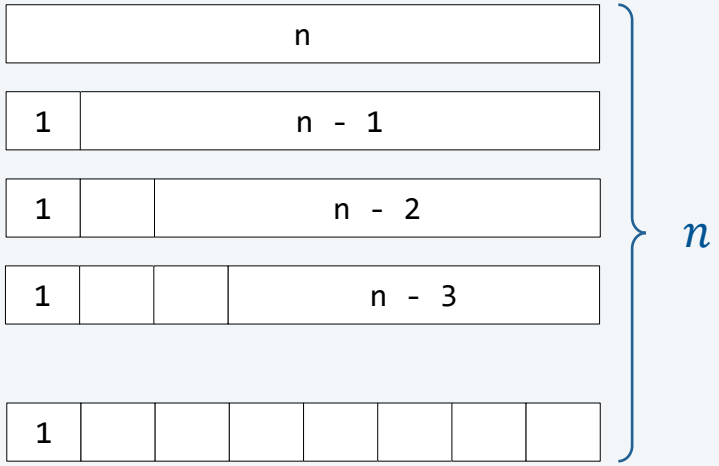
Gesamtanzahl der erforderlichen Vergleiche:

$$\approx n \cdot log_2\,(n)$$

ergibt

$$O\big(n \cdot log_2(n)\big)$$

Ungünstiger Fall:



- Bei jedem Durchlauf $n - i$ Vergleiche
- Durchläufe: n

Gesamtanzahl der erforderlichen Vergleiche:

$$n \cdot n$$

ergibt

$$O(n^2)$$

Analyse des Laufzeitverhaltens

Die quadratische Laufzeitkomplexität des `QuickSort` im ungünstigsten Fall ist sein prinzipieller Makel (dieser Fall tritt allerdings so gut wie nie auf)

Im durchschnittlichen Fall sind $\approx 1.4 \cdot n \cdot log_2(n)$ Vergleiche erforderlich (empirisch nachgewiesen), der `QuickSort` hat also im durchschnittlichen Fall die asympt. Laufzeitkompexität $O\big(n \cdot log_2(n)\big)$

Der `QuickSort` gehört damit zu den besten Sortierverfahren

9.5.1 Optimierung

Optimierungsidee

Wir verwenden das für kleine Felder (n < 10) hinsichtlich seines Laufzeitverhaltens günstige Einfügesortierverfahren

```
QuickSort(↓↑a: ArrayType ↓first: int ↓last: int)
  var
    x: KeyType
    i, j: int
begin
  if (last - first) < 10 then
    InsertionSort(↓↑a ↓first ↓last)
    return
  end -- if
  …
```

9.6 Das Mergesort-Verfahren

John von Neumann hat 1945 Mergesort erfunden/entwickelt und auf Computern ausgeführt

Der Algorithmus beruht auf der Strategie Teile und Herrsche

Lösungsidee: Das Feld `a[first:last]` mit `first < last` soll sortiert werden:

- Teile a in 2 gleich große Teilfelder `a[first:mid]` und `a[mid + 1:last]`
- Sortiere das Teilfeld `a[first:mid]` mittels Mergesort
- Sortiere das Teilfeld `a[mid + 1:last]` mittels Mergesort
- Mische die beiden sortierten Teilfelder sortiert zusammen (*merge*)

Grobstruktur und Gegenüberstellung

Grobstruktur von Mergesort

```
MergeSort(↓↑a ↓first ↓last)
begin
  if first < last then
    mid := (first + last) div 2
    MergeSort(↓↑a ↓first ↓mid)
    MergeSort(↓↑a ↓mid + 1 ↓last)
    Merge(↓↑a ↓first ↓mid ↓last)
  end -- if
end -- MergeSort
```

gleich große Teilfelder

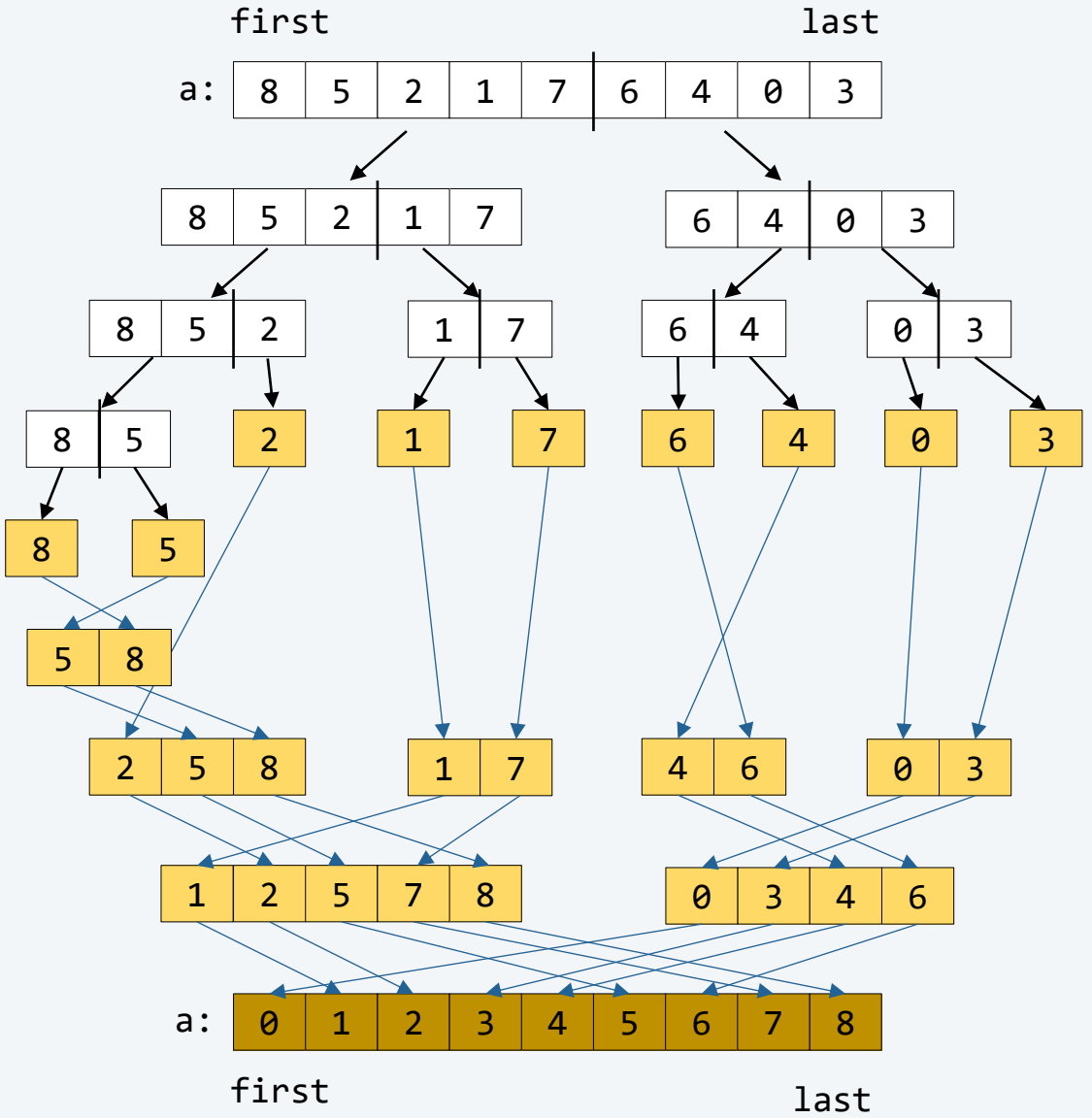
```
QuickSort(↓↑a ↓first ↓last)
begin
  if first < last then
    x := …
    Partition(↓a ↓x ↓first ↓last ↑i ↓↑j)
    QuickSort(↓↑a ↓first ↓j)
    QuickSort(↓↑a ↓i ↓last)
  end -- if
end -- QuickSort
```

Teilung hängt von x ab

MergeSort arbeitet „gegensätzlich“ zu Quicksort

- Quicksort erledigt die eigentliche Sortierarbeit bereits im Zuge der Teilung des Felds durch Umordnen der Elemente (oben: Partition)
- bei Mergesort wird im Zuge der Teilung des Felds noch keine Sortierarbeit geleistet
- erst nach den beiden rekursiven Aufrufen werden die sortierten Teilfelder zu einem sortierten Gesamtfeld zusammengemischt

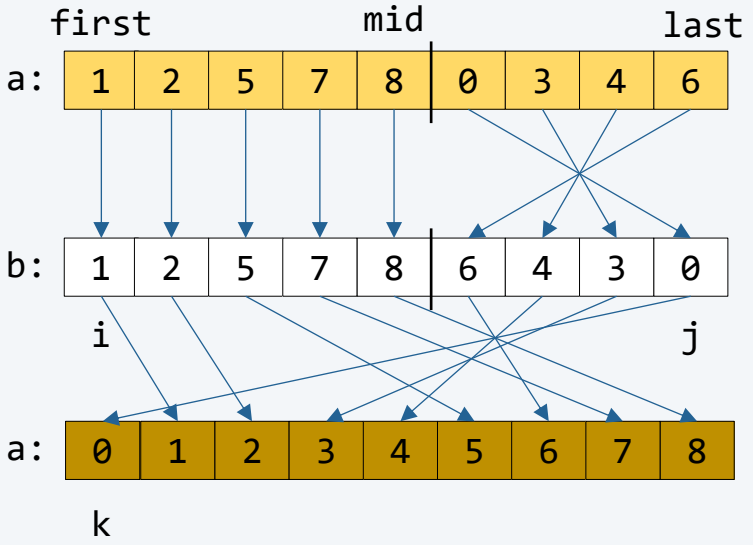
Ablauf des Mergesort-Verfahrens



Teilfelder sortiert zusammenmischen (merge)

Mergesort erfordert das Zusammenfügen von zwei sortierten Teilfeldern zu einem sortierten Teilfeld

Dazu wird ein Hilfsfeld b der Länge n benötigt (mit Optimierung nur Länge n/2)



Merge:

```
i := first; j := last
for k := first to last do
  if b[i].key < b[j].key then
    a[k] := b[i]; i := i + 1
  else
    a[k] := b[j]; j := j - 1
  end -- if
end -- for
```

Algorithmus MergeSort

```
MergeSort(↓↑a: ArrayType ↓first: int ↓last: int)
  var
    i, j, k, mid: int
begin
  if first < last then
    mid := (first + last) div 2
    MergeSort(↓↑a ↓first ↓mid)
    MergeSort(↓↑a ↓mid + 1 ↓last)
    for i := mid downto first do
      b[i] := a[i] end
    for j := mid + 1 to last do
      b[last + mid + 1 - j] := a[j] end
    i := first; j := last
    for k := first to last do
      if b[i].key < b[j].key then
        a[k] := b[i]; i := i + 1
      else
        a[k] := b[j]; j := j - 1
      end -- if
    end -- for
  end -- if
end MergeSort
```

```
var
  b: ArrayType
```

Mergesort

- Wie der Quicksort hat Mergesort die asymptotische Laufzeitkomplexität von $O\left(n \cdot \log_2(n)\right)$
- Im Vergleich zu Quicksort kann Mergesort aber nicht degenerieren, also nicht quadratisch werden
- Es kommt für den durchschnittlichen Fall im Vergleich zu Quicksort mit weniger Vergleichen aus, benötig aber ein Hilfsfeld b der Länge n
- Das Mischsortieren ist attraktiv, wenn es um die Sortierung von Datenbeständen auf externen Speichermedien (z. B. Festplatten) geht

TimSort

- TimSort, entwickelt von Tim Peters für die Pgm.sprache Python, ist eine Erweiterung von MergeSort (und InsertionSort)
- TimSort zeichnet sich dadurch aus, dass er bei bereits teilweise vorsortierten Daten besonders effizient arbeitet

9.7 Weitere Sortierverfahren

Über die bisher vorgestellten Lösungsideen für das Sortierproblem existiert noch eine Vielzahl weiterer Vorschläge, von denen eine Auswahl hier noch skizziert wird (Details dazu finden sich im Buch)

Der **BubbleSort** (das so genannte Austauschsortierverfahren):

Eine Familie besonders einfacher Sortierverfahren beruht auf der Lösungsidee, dass so lange systematisch benachbarte Elemente miteinander verglichen und bei Bedarf vertauscht werden, bis das Feld sortiert ist. Da das Vertauschen benachbarter Elemente die zentrale Operation dieses Verfahren ist, nennt man es Austauschsortieren.

Die asymptotische Laufzeitkomplexität dieses Verfahrens ist $O(n^2)$

Weitere Sortierverfahren

Der **CombSort**

Dabei handelt es sich um eine Variante des Austauschsortierens.

Die Idee der Vorsortierung von Donald L. Shell kann in ähnlicher Weise und mit ebenso guten Ergebnissen auch auf das Austauschsortieren angewendet werden

Die engl. Bezeichnung CombSort für diesen Algorithmus leitet sich von comb (dt. Kamm) ab und soll andeuten, dass das zu sortierende Feld mit einem immer „feineren Kamm durchgekämmt“ wird. Die asymptotische Laufzeitkomplexität des CombSort ist ähnlich der des ShellSort $O(n^{1.3})$

Der **HeapSort**

John W. J. Williams und Robert W. Floyd haben 1964 mit ihren Ideen gemeinsam dazu beigetragen, dass auf Basis einer sogenannten Heap-Datenstruktur, ein effizientes Sortierverfahren entwickelt werden konnte, dessen Laufzeitverhalten zwar etwas schlechter als das des Quicksort von Hoare, aber dafür nicht mit dem Risiko einer quadratischen Laufzeitkomplexität (Degeneration), wie beim Quicksort der Fall, behaftet ist.

Weitere Sortierverfahren

Der **IndirectSort** (das sogenannte indirekte Sortieren)

Wir haben bei der Komplexitätsanalyse der Sortierverfahren darauf hingewiesen, dass neben der Anzahl der Schlüsselvergleiche auch die Anzahl der Zuweisung von Datenobjekten in der Regel eine nicht zu vernachlässigende Auswirkung auf das Laufzeitverhalten hat.

Sind die zu sortierenden Datenobjekte sehr groß, ist es zweckmäßig, den Aufwand für das Verschieben der Datenobjekte zu minimieren. Das kann man durch indirektes Sortieren erreichen. Dabei wird nicht der Datenbestand selbst sortiert, sondern ein Feld von Zeigern, die auf die entsprechenden Datenobjekte verweisen.

Weitere Sortierverfahren

Der **BucketSort** (das sogenannte Fächersortieren)

Unter gewissen Voraussetzungen (Einschränkungen) ist das Sortieren sogar in linearer Zeit, also mit einer asymptotischen Laufzeitkomplexität $O(n)$ möglich.

Wenn z. B. die Schlüsselwerte aus einem relativ kleinen numerischen Bereich `1:max` stammen, kann man ein Hilfsfeld `h` mit `max` Elementen verwenden, in dem verkettete Listen aus Datenobjekten so verankert werden, dass jedes Datenobjekt unter Heranziehung seines Schlüsselwerts `x` als Index im Feld `h` in die entsprechende Liste `h[x]` eingefügt wird. Ein abschließender Durchlauf durch das Feld `h` und durch die darin verankerten Listen ermöglicht es, die Datenobjekte in eine sortierte Reihenfolge zu bringen. Das Feld `h` kann als „Schrank mit Fächern“ aufgefasst werden, in welche die entsprechenden Datenobjekte einsortiert werden. Deshalb wird dieses Verfahren auch als Fächersortieren (BucketSort) bezeichnet.

9.8 Stabilität von Sortierverfahren

Ein Sortierverfahren wird als stabil (*stable*) bezeichnet, wenn die relative Reihenfolge von Datenobjekten mit gleichem Schlüsselwert durch den Sortiervorgang unverändert bleibt.

Stabilität ist eine Eigenschaft, die nur wenige Sortierverfahren aufweisen, die aber für bestimmte Anwendungen essenziell ist.

Beispiele für instabile Sortierverfahren:

- Selectionsort, Shell-Sortieren, Combsort, Quicksort

Beispiele für stabile Sortierverfahren:

- Insertionsort, Bubblesort, Mergesort

9.9 Das topologische Sortieren

Problemstellungen:

- Nach einem Studienplan müssen gewisse Kurse vor anderen besucht werden, da sich manche Kurse im Stoff auf früher gelehrte Voraussetzungen stützen. Ist ein Kurs v Voraussetzung für einen Kurs w, so schreiben wir $v \ll w$ (v „vor“ w).
  Topologisches Sortieren bedeutet das Anordnen der Kurse so, dass kein Kurs einen später aufgeführten Kurs voraussetzt.
- Ein Prozess (z. B. ein technisches Projekt) wird in Teilprozesse aufgeteilt. Gewisse Teilprozesse müssen gewöhnlich abgeschlossen sein, bevor andere Teilprozesse in Angriff genommen werden können. Ist ein Teilprozess v vor einem Teilprozess w auszuführen, so schreiben wir $v \ll w$ (v „vor“ w).
  Topologisches Sortieren bedeutet eine Anordnung derart zu finden, dass bei Inangriffnahme jedes Teilprozesses alle vorbedingten Teilprozesse bereits erledigt sind.

Problemstellung

Gegeben ist also eine Menge S von Elementen auf der eine teilweise Ordnung definiert ist

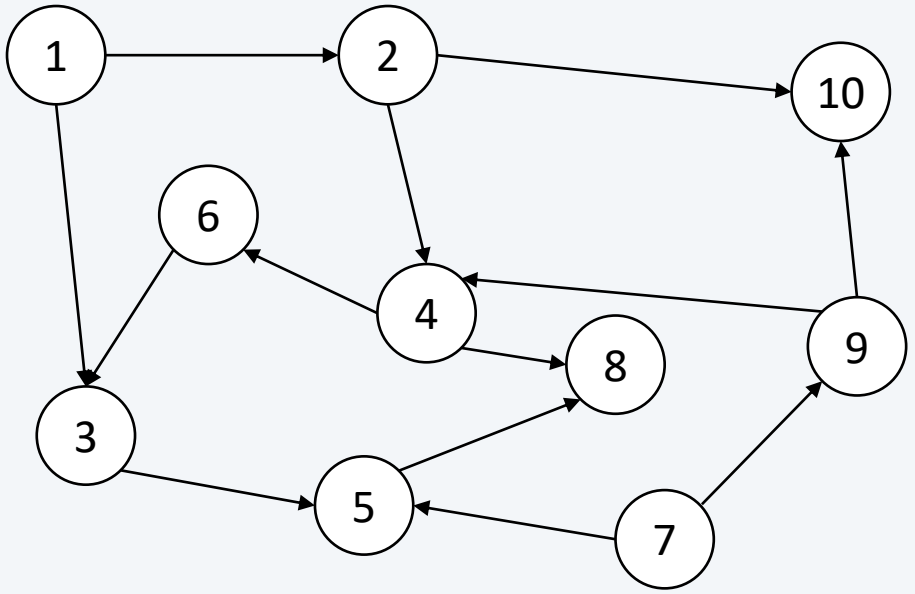
Die teilweise Ordnung wird durch die Relation $\ll$ (in Worten: geht voran) gebildet und diese Relation erfüllt folgende drei Eigenschaften für beliebige $x, y, z \in S$

1. Wenn $x \ll y$ und $y \ll z$ gilt, dann gilt $x \ll z$ (Transitivität)
2. Wenn $x \ll y$ gilt, dann gilt nicht $y \ll x$ (Asymmetrie)
3. Es gilt nicht $x \ll x$ (Irreflexibilität)

Aus verständlichen Gründen nehmen wir an, dass die Menge S, die topologisch sortiert werden soll, endlich ist

Problemstellung

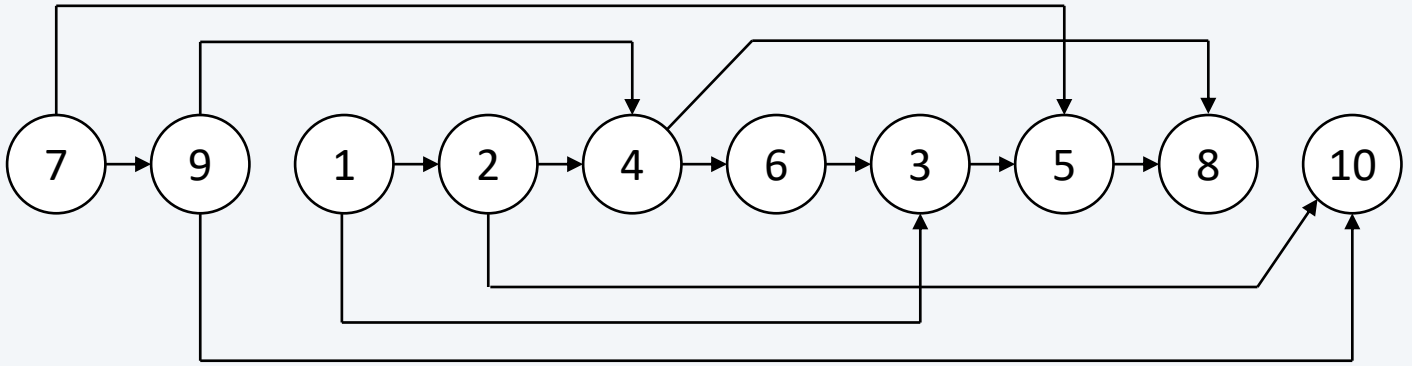
Teilweise Ordnungen können wir als gerichtete Graphen darstellen, in denen die Knoten die Elemente aus S und die gerichteten Kanten die Ordnungsrelation darstellen, z. B.



partiell geordnete Menge

Problemstellung

Das Problem des topologischen Sortierens besteht darin, die teilweise Ordnung in eine lineare Ordnung einzubetten, z. B.



Die Eigenschaften (1) und (2), Transitivität und Asymmetrie garantieren, dass der Graph keine Zyklen enthält (genau unter dieser Voraussetzung ist eine solche Transformation überhaupt möglich)

Lösungsidee

Lösungsidee für das topologische Sortieren

- Wir wählen ein Element aus S das keinen Vorgänger hat; dieses Element kommt an den Anfang der Ergebnisfolge und wird aus S entfernt
- Die restliche Menge $S´$ ist immer noch partiell geordnet und wir verfahren mit ihr in der gleichen Weise, d. h. ein Element ohne Vorgänger wird ausgewählt und an die bereits existierende Ergebnisfolge angefügt, bis $S´$ leer geworden ist

Lösungsidee

Lösungsidee für die Wahl der Datenstrukturen

- Die Wahl der Datenstruktur wird durch die darauf auszuführenden Operationen bestimmt, im vorliegenden Falle speziell durch die Operation des Auswählens eines Elementes ohne Vorgänger und seine Entfernung aus S
- Jedes Element soll deshalb folgende vier Komponenten haben

  (1) Objektwert (Sortierschlüssel; z. B. Teilprozessnummer),

  (2) die Anzahl seiner Vorgänger,

  (3) die Menge seiner Nachfolger,

  und weil die Anzahl der Elemente von S beliebig sein kann, realisieren wir die Menge S als vernetztes Datenobjekt in Form einer einfach-verketteten Liste und benötigen daher noch

  (4) ein Verbindungselement zum Aufbau der Liste (Menge S)

Lösungsidee

Lösungsidee für die Wahl der Datenstrukturen

Da auch die Menge der Nachfolger jedes Elements unbekannt (und beliebig) ist, realisieren wir diese ebenfalls in Form einer einfach verketteten Liste

Wenn wir die Knoten der Hauptliste, in der jedes Element von S einmal vorkommt mit dem Typ `Leader` und die Elemente der Liste der Nachfolger mit dem Typ `Trailer` modellieren, so erhalten wir:

```
type
  LeaderPtr = →Leader
  TrailerPtr = →Trailer
  Leader = compound
    val, count: int
    next: LeaderPtr
    trailer: TrailerPtr
   end -- Leader
   Trailer = compound
     id: LeaderPtr
     next: TrailerPtr
   end -- Trailer
```

Lösungsidee

Lösungsidee für den Algorithmus zum Aufbau der Datenstruktur für S

- Zur Verallgemeinerung nehmen wir an, die Ordnungsrelation sei als Folge von Paaren x, y mit der Bedeutung $x \ll y$ auf einem sequentiellen Eingabemedium verfügbar
- Der Aufbau der Datenstruktur zur Repräsentation von *S* erfolgt so:
  1. Fortgesetztes Lesen der Relationspaare x, y
  2. Die zu x und y gehörenden Listenknoten müssen zuerst in der `Leader`-Liste gesucht und, falls noch nicht vorhanden, eingefügt werden. Dazu verwenden wir einen Funktionsalgorithmus `Find` (mit `p := Find(`$\downarrow$`x)`), der als Resultat einen Zeiger auf den entsprechenden Knoten liefert
  3. Einfügen eines neuen Knotens in die Liste der `Trailer` von `x`, versehen mit der Identifikation von y
  4. Erhöhung des Zählers der Vorgänger von y um 1

Transformation der Lösungsidee in einen Algorithmus

```
interface of TopSort
  CreateDataStructure()
  TopSort()
end TopSort
```

```
implementation of TopSort
  type
    LeaderPtr = →Leader
    TrailerPtr = → Trailer
    Leader = compound
      val, count: int
      next: LeaderPtr
      trailer : TrailerPtr
    end -- Leader
    Trailer = compound
      id: LeaderPtr
      next: TrailerPtr
    end -- Trailer
  var
    head, tail: LeaderPtr
    n: int
```

repräsentiert die zu sortierende Menge

Anzahl der Elemente der zu sortierenden Menge

Transformation der Lösungsidee in einen Algorithmus

```
CreateDataStructure()
  var
    done: bool
    x, y: int
    p, q: LeaderPtr
    t: TrailerPtr
begin
  head := New(↓Leader); tail := head; n := 0
  Read(↑x ↑done)
  while done do
    Read(↑y ↑done)
    p := Find(↓x)
    q := Find(↓y)
    t := New(↓Trailer); t→id := q
    t→next := p→trailer; p→trailer := t
    q→count := q→count + 1
    Read(↑x ↑done)
  end -- while
end CreateDataStructure
```

Lösungsidee

Lösungsidee für den Algorithmus zum topologischen Sortieren von *S*

- Da der Prozess des topologischen Sortierens aus dem fortgesetzten Auswählen eines Knotens besteht, dessen Anzahl der Vorgänger null ist, ist es sinnvoll, alle diese Knoten zu einer Liste zusammenzufassen
- Da wir die ursprüngliche Liste der `Leader-Knoten` nicht mehr benötigen, können wir die Komponente `next` verwenden, um die Knoten ohne Vorgänger miteinander zu verbinden
- Aus Gründen der Einfachheit, erstellen wir diese Liste in „umgekehrter Reihenfolge“
- Erst nach dieser Vorbereitung kommen wir zum eigentlichen Problem des topologischen Sortierens

Lösungsidee

Lösungsidee für den Algorithmus zum topologischen Sortieren von *S*

Das topologische Sortieren von S, d. h. das Erstellen (Ausgeben) der Ergebnisfolge kann dann in folgender Weise erfolgen:

1. gib den Inhalt der Knoten (Komponente `val`) der Liste aller Knoten ohne Vorgänger der Reihe nach aus (bis die Liste leer geworden ist)
2. vermindere die Zähler der Vorgänger für jeden der Nachfolger in der Liste der Trailer des ausgegebenen Knotens und prüfe, ob dabei ein Zähler 0 wird; wenn ja, dann nimm diesen Knoten in die Liste der Konten ohne Vorgänger auf

Transformation der Lösungsidee in den Algorithmus TopSort

```
TopSort()
  var
    p, q: LeaderPtr
    t: TrailerPtr
  begin
    p := head
    head := null
    while p ≠ tail do
      q := p
      p := q→next
      if q→count = 0 then
        q→next := head
        head := q
      end -- if
    end -- while
    -- …
```

Suche und verbinde alle Knoten ohne Vorgänger

Knoten an Ergebnisliste anfügen

head identifiziert die Ergebnisliste

Transformation der Lösungsidee in den Algorithmus TopSort

```
    -- Gib die Knoteninhalte (val-Komponente) in
    -- topologisch sortierter Reihenfolge aus
    q := head
    while q ≠ null do
      Write(↓q→val); n := n – 1; t := q→trailer;
      q := q→next
      while t ≠ null do
        p := t→id;  p→count := p→count – 1
        if p→count = 0 then
          p→next := q; q := p
        end -- if
        t := t→next
      end -- while
    end -- while

    if n ≠ 0 then
      Write("Menge ist nicht partiell geordnet")
    end -- if
  end TopSort
```

q zeigt auf den Anfang der Ergebnisliste

füge p in Ergebnisliste ein

Prüfung ob alle Knoten ausgegeben sind

Vervollständigung des Moduls TopSort

Bleibt noch der Algorithmus `Find` (zur Suche eines Knotens in der `Leader`-Liste) zu formulieren:

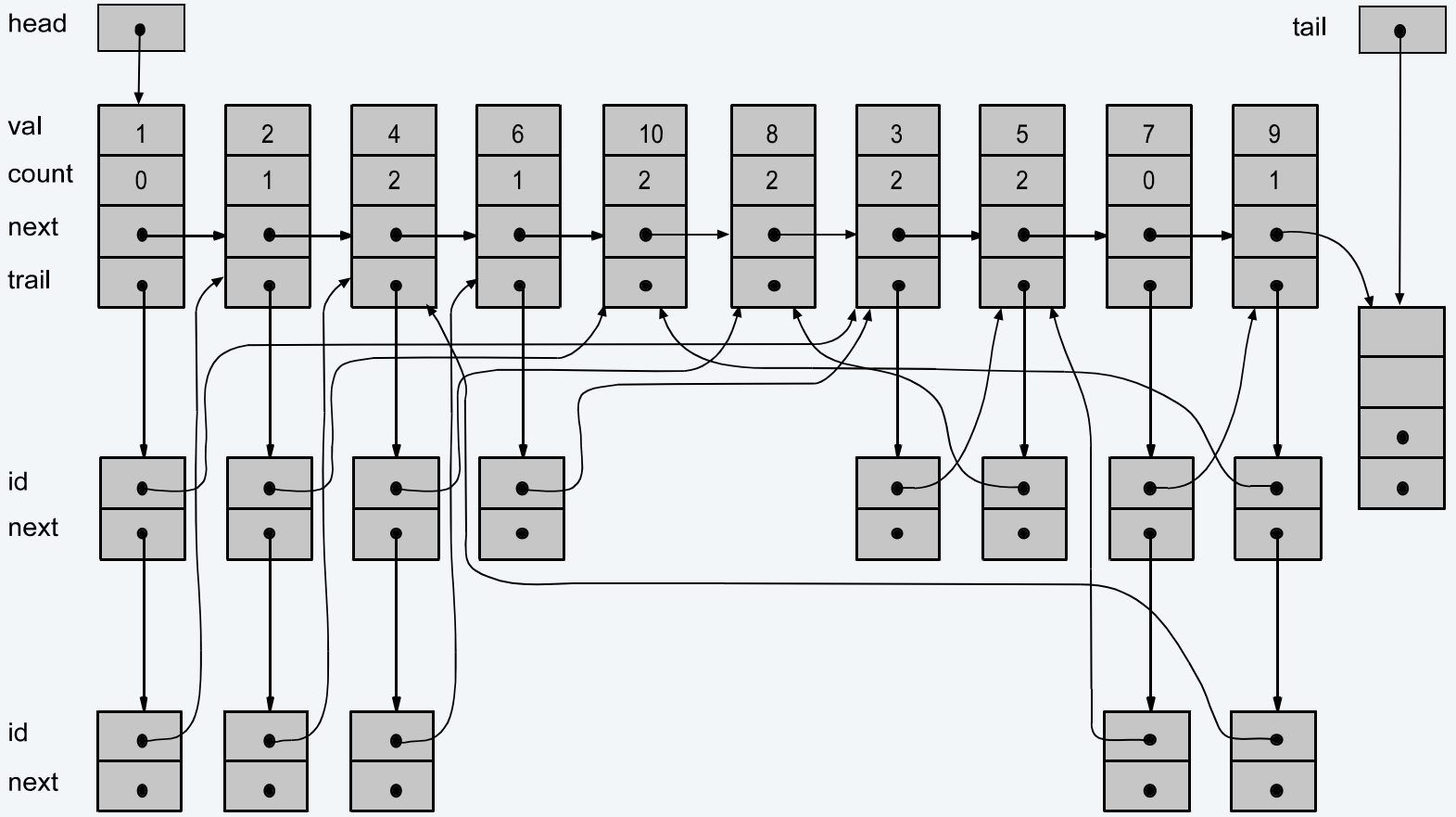
```
Find (↓w: int): LeaderPtr
  var
  h: LeaderPtr
begin
  h := head;  tail→val := w -- sentinel
  while h→val ≠ w do
    h := h→next
  end -- while
  if h = tail then
    tail := New(↓Leader); n := n + 1
    h→count := 0;  h→trailer := null;  h→next := tail
  end -- if
  return h
end Find
```

und die Standardinitialisierung im Modulrumpf vorzunehmen:

```
begin
  head, tail := null; n := 0
end TopSort
```

Aufbau der Datenstruktur für das obige Beispiel

Zustand nach Ausführung von `CreateDataStructure`



Beim topologischen Sortieren
erzeugte Listenstruktur

Topologisches Sortieren

Wir haben eine Gedächtnisvariable n verwendet, mit der beim Aufbau der Datenstruktur für die zu sortierende Menge, die Anzahl der Knoten gezählt werden. Dieser Zähler wird bei der topologisch sortierten Ausgabe mit jedem ausgegebenen Knotenwert um 1 vermindert. Am Ende muss n 0 sein, sonst sind nicht alle Knotenwerte ausgegeben (in diesem Fall ist die Menge *S* offensichtlich nicht partiell geordnet)

Die hier verwendete Lösungsidee stammt von N. Wirth.