Teil 3 der Grundvorlesung

3. Grundlegende Verfahren

~~3.1 Speicherverwaltung~~
~~3.2 Suchverfahren~~
~~3.3 Hashing~~
3.4 Sortieren
3.5 Algorithmen auf Graphen
3.6 Zufallszahlen

Gliederung des Kapitels

3.4 Sortieren

3.4.1 Vorbemerkungen
3.4.2 Sortieren durch Aussuchen/Auswählen
3.4.3 Sortieren durch Einfügen
3.4.4 Sortieren durch Austauschen
3.4.5 Mischen (meist für externes Sortieren)
3.4.6 Streuen und Sammeln
3.4.7 Paralleles Sortieren

3.4.1 Vorbemerkungen

Suchen und Sortieren machen einen Großteil aller Verwaltungstätigkeiten aus. Dies gilt insbesondere im Rechner, wo Daten ständig abgelegt und schnell wiedergefunden werden müssen. Das Sortieren nutzt die Anordnung der Schlüsselmenge direkt aus und ermöglicht ein schnelles Auffinden: O(log(n)) durch Intervallsuche oder O(log(log(n))) durch Interpolationssuche, siehe 3.2.1.

Wir klären als erstes den Begriff "Sortieren" und leiten dann eine *untere* Schranke für die Zahl der Vergleiche her, die bei einem Sortierverfahren, das ausschließlich auf Vergleichen basiert, erforderlich sind. Sodann geben wir einen Überblick über gängige Sortierverfahren und ihre Komplexität.

Definition 3.4.1.1: Sortierte Folgen

Gegeben sei eine endliche oder unendliche Menge mit totaler Ordnung $A = \{a_1, ..., a_s\}$ oder $A = \{a_1, a_2, a_3, a_4, ...\}$ mit $a_1 < a_2 < a_3 < a_4 < ...$.

(1) Eine Folge $v = v_1 v_2 ... v_n$ mit $v_i \in A$ (d.h., $v \in A^*$) heißt (aufsteigend) geordnet genau dann, wenn gilt $v_1 \le v_2 \le ... \le v_n$.
(Speziell ist jede leere oder einelementige Folge geordnet.)

(2) Eine Folge $v = v_1 v_2 ... v_n$ mit $v_i \in A$ (d.h., $v \in A^*$) heißt invers oder absteigend geordnet $\Leftrightarrow$ $v_n \le v_{n-1} \le ... \le v_1$.
(Speziell ist jede leere oder einelementige Folge invers geordnet.)

Die Sortieraufgabe lautet dann: Ordne eine Folge von n Elementen so um, dass sie sortiert ist. Wir präzisieren dies nun.

Definition 3.4.1.2: Permutationen

Es sei n eine natürliche Zahl.
Eine bijektive Abbildung $\pi: \{1, 2, ..., n\} \to \{1, 2, ..., n\}$ heißt Permutation der Ordnung n.
[bijektiv = injektiv und surjektiv, d.h.,
injektiv: zu je zwei Elementen $i \neq j$ gilt $\pi(i) \neq \pi(j)$ und
surjektiv: zu jedem j existiert ein i mit $\pi(i) = j$.
Eine Permutation ist also nur eine Umordnung der n Elemente.]

Hinweis: Bekanntlich gibt es genau n! verschiedene Permutationen der Ordnung n.

Definition 3.4.1.3: Die Sortieraufgabe lautet:
Finde zu einer beliebigen Folge $v = v_1 v_2 ... v_n \in A^*$ eine Permutation π der Ordnung n mit: $v_{\pi(1)} v_{\pi(2)} ... v_{\pi(n)}$ ist sortiert (oder invers sortiert, je nach Anwendung).

Meist hängt ein Sortieralgorithmus ab von Fragen wie:

- Wie sind die Daten gegeben, zu welchen Mengen gehören sie?
- Wo liegen die Daten? (Hauptspeicher, Platten, Bänder, ...?)
- Zulässige Operationen? (Vertauschen ...?)
- Behandlung gleicher Elemente? ($\Rightarrow$ Stabilität.)
- Nur Zugriffsstruktur oder Gesamtdaten sortieren?
- Gibt es zusätzlichen Speicher oder nicht?
- Sequentielles, paralleles, verteiltes Sortieren?
- Effizienz im Mittel oder auch im worst case?
- Erkennen oder Beachten von Vorsortierungen?

3.4.1.4: Jedes Element v_i der zu sortierenden Folge $v = v_1v_2...v_n$ ist in der Praxis meist ein umfangreicher Datensatz (record). Die Folge wird in der Regel nur nach einem Ordnungskriterium sortiert.

Fall 1: Dieses Kriterium wird durch eine Funktion g: A → M beschrieben, wobei M eine geordnete Menge ist (A braucht in diesem Fall gar nicht sortierbar zu sein). $g(v_i) = k_i$ heißt dann der Schlüssel von v_i, der in der Regel im record v_i enthalten ist. (Wir können also stets Fall 2 annehmen.)

Fall 2: Das Ordnungskriterium bezieht sich auf eine oder mehrere Komponenten des records. Den Vektor dieser Komponenten, nach denen zu sortieren ist, bezeichnen wir als Schlüssel. Man verlangt oft, dass dieser ein Element eindeutig beschreibt, doch lassen wir hier auch verschiedene Elemente mit gleichem Schlüssel zu.

In der Regel sortiert man nur die Schlüssel einer Folge.

Beispielsweise wird folgende Folge aus Name und Alter
(Beier, 23), (Zahn, 40), (Fuhr, 30), (Horn, 41), (Beier, 30), (Horn, 17)
zur Folge
(Beier, 23), (Beier, 30), (Fuhr, 30), (Horn, 41), (Horn, 17), (Zahn, 40),
sofern nach dem Namen sortiert wird.
Dagegen erhält man die Folge
(Horn, 17), (Beier, 23), (Beier, 30), (Fuhr, 30), (Zahn, 40), (Horn, 41),
falls nach dem Alter sortiert wird.

Man erkennt, dass eine Wahlfreiheit vorliegt, wenn *gleiche Schlüssel* auftreten. Statt der Reihenfolge
(Beier, 23), (Beier, 30), (Fuhr, 30), (Horn, 41), (Horn, 17), (Zahn, 40)
hätte man beim Sortieren nach dem Namen auch die Folge
(Beier, 30), (Beier, 23), (Fuhr, 30), (Horn, 17), (Horn, 41), (Zahn, 40)
als korrektes Ergebnis erhalten können. In der Regel möchte man die Vorsortierung aufrecht erhalten.

Wird also eine Folge v nach mehreren Komponenten (Schlüsseln) nacheinander geordnet, so kann man die Folge zunächst nach dem am wenigsten relevanten Kriterium (im Beispiel: nach dem Alter) und dann schrittweise nach dem nächstwichtigeren Kriterium sortieren. Hierfür muss ein Sortierverfahren "stabil" sein.

Definition 3.4.1.5:

Ein Sortierverfahren *Sort* heißt **stabil**, wenn *Sort* die Reihenfolge von Elementen mit gleichem Schlüssel nicht verändert, d.h.: wenn $Sort(v_1v_2...v_n) = v_{\pi(1)}v_{\pi(2)}...v_{\pi(n)}$ die Sortierabbildung ist, dann gilt für alle $v_{\pi(i)} = v_{\pi(j)}$ mit $\pi(i) < \pi(j)$ stets $i < j$.

Sort heißt invers stabil, wenn die Reihenfolge der Elemente mit gleichem Schlüssel von *Sort* gespiegelt wird.

Beispiel: Ein stabiles Sortierverfahren wird aus der Folge
(Beier, 23), (Zahn, 40), (Fuhr, 30), (Horn, 41), (Beier, 30), (Horn, 17)
beim Sortieren nach dem Alter die Folge
(Horn, 17), (Beier, 23), (Fuhr, 30), (Beier, 30), (Zahn, 40), (Horn, 41)
liefern und das anschließende Sortieren nach dem Namen ergibt:
(Beier, 23), (Beier, 30), (Fuhr, 30), (Horn, 17), (Horn, 41), (Zahn, 40).

Wir erhalten also die Reihenfolge, die wir erwarten würden: Die Liste ist nach dem Namen sortiert und gleiche Namen sind aufsteigend nach dem Alter angeordnet.

Wir haben bereits einige Sortierverfahren kennen gelernt:

In 1.4.4.4: Sortieren durch Austauschen benachbarter, falsch stehender Elemente (*Bubble Sort*).

In 1.6.4.4: *Sortieren mit Bäumen,* indem die Glieder der Folge nacheinander in einen binären Suchbaum eingefügt und anschließend in inorder-Reihenfolge ausgelesen werden.

1.7.3.3: *Quicksort.* Man wählt ein "Pivot"-Element p aus und spaltet die in einem array gespeicherte Folge in zwei Teilfolgen, von denen alle Elemente der ersten Teilfolge kleiner oder gleich p und alle Elemente der zweiten Teilfolge größer oder gleich p sind. Danach: rekursiv weiter mit beiden Teilfolgen, sofern die jeweilige Teilfolge noch mindestens zwei Elemente besitzt.

Frage an Sie: Welche dieser drei Verfahren sind stabil?

Sortieren erfordert in der Praxis viele Umspeicherungsoperationen. Sind die Elemente sehr groß, so kostet dies viel Zeit. Man zieht daher die Schlüssel und den Verweis auf den jeweiligen Datensatz heraus und sortiert nur diese Schlüssel-Verweis-Tabelle. Wir werden also unseren Sortierverfahren Elemente des Datentyps

```
record  key: <Typ des Schlüssels>;
        zeiger: <Zeigertyp auf den Elementtyp>;
end record
```

zugrunde legen. Es genügt, sich nur auf die Sortierung der Schlüssel zu beschränken.

Definition 3.4.1.6:

Für eine Permutation $\pi: \{1, 2, ..., n\} \to \{1, 2, ..., n\}$ heißt

$$I(\pi) = \left| \{ (i,j) \mid i < j \text{ und } \pi(i) > \pi(j) \} \right|$$

die Inversionszahl (oder der Fehlstand) von π.

Analog: Für eine Folge $v_1 v_2 ... v_n \in A^*$ heißt

$$I(v) = \left| \{ (i,j) \mid i < j \text{ und } v_i > v_j \} \right|$$

die Inversionszahl (oder der Fehlstand) von v.

Wegen $I(v_1 v_2 ... v_n) + I(v_n v_{n-1} ... v_1) = \left| \{ (i,j) \mid i < j \} \right|$ gilt der

Hilfssatz 3.4.1.7: $I(v_1 v_2 ... v_n) + I(v_n v_{n-1} ... v_1) = ½ \cdot n \cdot (n-1)$.

Wegen $I(1\ 2\ 3\ ...\ n) = 0$ folgt $I(n\ n\text{-}1\ ...2\ 1) = ½ \cdot n \cdot (n-1)$.

Definition 3.4.1.8:

Ein Sortierverfahren *Sort* heißt ordnungsverträglich ⇔
Je geordneter die zu sortierende Folge bereits ist, umso schneller arbeitet *Sort*.

Genauer:
Wenn *Sort* für zwei Folgen $v=v_1 v_2 ... v_n$ und $w=w_1 w_2 ... w_n$ die Permutationen π_1 und π_2 realisiert und wenn die Inversionszahl von π_1 kleiner als die von π_2 ist, dann ist auch die Zeit, die *Sort* zum Sortieren von v benötigt, kleiner als die Zeit zum Sortieren von w.

Wir betrachten die Operation "benachbartes Vertauschen". Diese überführt eine Folge

$$v_1\ v_2 ...\ v_{i-1}\ v_i\ v_{i+1}\ v_{i+2} ...\ v_n$$

in die Folge

$$v_1\ v_2 ...\ v_{i-1}\ v_{i+1}\ v_i\ v_{i+2} ...\ v_n \quad \text{(für ein } 0<i<n\text{)}.$$

Hierfür gilt:

$$\left| I(v_1 v_2 ... v_{i-1} v_i v_{i+1} v_{i+2} ... v_n) - I(v_1 v_2 ... v_{i-1} v_{i+1} v_i v_{i+2} ... v_n) \right| = 1.$$

Somit haben wir gezeigt:

Hilfssatz 3.4.1.9:
Ein Sortierverfahren, das ausschließlich mit der Operation "benachbartes Vertauschen" arbeitet, benötigt im worst case mindestens $½ \cdot n \cdot (n-1)$ Schritte.

Wir betrachten die Operation "Vertauschen". Diese überführt eine Folge (für $1 \le i \le j \le n$)

$$v_1\ v_2 ...\ v_{i-1}\ v_i\ v_{i+1} ...\ v_{j-1}\ v_j\ v_{j+1} ...\ v_n$$

in die Folge

$$v_1\ v_2 ...\ v_{i-1}\ v_j\ v_{i+1} ...\ v_{j-1}\ v_i\ v_{j+1} ...\ v_n .$$

Hierfür gilt:

$$0 \le \left| I(v_1 ... v_i ... v_j ... v_n) - I(v_1 ... v_j ... v_i ... v_n) \right| \le n\text{-}1,$$

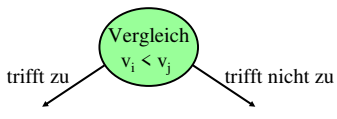
wobei der größte Wert n-1 nur erreicht wird, wenn das kleinste Element am Ende stand und durch die Vertauschung an den Anfang gebracht wurde bzw. eine hierzu symmetrische Situation vorliegt. Es folgt:

Hilfssatz 3.4.1.10: Ein Sortierverfahren, das ausschließlich mit der Operation "Vertauschen" arbeitet, benötigt im worst case mindestens $½ \cdot n$ Schritte.

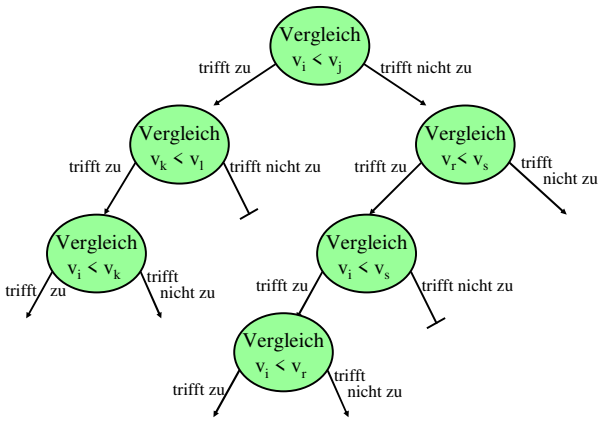
In der Regel weiß man nicht, ob man zwei Elemente einer Folge vertauschen soll. Diese Entscheidung wird durch einen Vergleich getroffen: Falls $v_i < v_j$ und $i > j$ ist, dann vertauscht man diese beiden Elemente in der Folge.

Wird das Vertauschen durch vorher gehende Vergleiche gesteuert, so dauert das Sortierverfahren mindestens so lange wie die Anzahl der hierfür erforderlichen Vergleiche.

Eine Folge von Vergleichen bildet einen binären Baum, der aus den Knoten und Kanten



aufgebaut ist.



Hat man alle Informationen zum Sortieren gewonnen, dann stellen die null-Zeiger in diesem Baum die Permutationen dar, die zur Sortierung gehören. Da es n! Permutationen der Ordnung n gibt, muss der "Baum der Vergleiche" daher mindestens n! null-Zeiger besitzen.

Ein binärer Baum mit m-1 Knoten besitzt genau m null-Zeiger. Also muss der Baum der Vergleiche
mindestens n!-1
Knoten besitzen.
Die Länge des längsten Weges, also die Tiefe dieses Baums gibt die Zahl der erforderlichen Vergleiche im worst case an. Die Tiefe eines binären Baums mit k Knoten ist aber mindestens log(k+1), siehe Folgerung 3.2.2.12.

Satz 3.4.1.11: Ein Sortierverfahren, das ausschließlich auf Vergleichen zweier Elemente beruht, benötigt im worst case
mindestens $\log(n!) \approx n \cdot \log(n) - 1{,}4404 \cdot n$
Schritte.

Hinweis: Wende die Stirlingschen Formel an: Zu jedem n gibt es ein d mit 0<d<1, so dass gilt (mit e=2,718281828...):

$$n! = \left(\frac{n}{e}\right)^n \sqrt{2\pi n}\ e^{\frac{1}{12}d}$$

Durch Logarithmieren erhält man hieraus:
$\log(n!) \approx n \cdot \log(n) - n \cdot \log(e) \approx n \cdot \log(n) - 1{,}4404 \cdot n$

3.4.1.12 Überblick über die üblichen Sortiermethoden:

Aussuchen / Auswählen:
- a. Minimumsuche (minimum sort)
- b. Heapsort (normal, bottom up, ultimativ)

Einfügen:
- a. Einfügen in Listen (Insertion sort)
- b. Baumsortieren (mit binären Bäumen, AVL-Bäumen, ...)
- c. Fachverteilen (und radix exchance)

Austauschen:
- a. Benachbartes Austauschen (bubble sort, shaker sort)
- b. Shellsort
- c. Quicksort

Mischen:
- merge sort und diverse Varianten

Streuen und Sammeln (bucket sort)

Sortiermethoden	Zeitaufwand	zusätzl. Platz
Aussuchen / Auswählen		
a. Minimumsuche	$\frac{1}{2} \cdot n^2$	konstant
b. Heapsort	$\leq 2n \cdot \log(n)$	konstant
Einfügen		
a. Einfügen in Listen	$\frac{1}{2} \cdot n^2$	konstant
b. Baumsortieren (AVL-Bäume)	$\leq 1{,}4404 \cdot n \cdot \log(n)$	O(n)
c. Fachverteilen (im Mittel)	$O(n \cdot \log(n))$	O(n)
Austauschen		
a. Benachb. Austauschen	$\frac{1}{2} \cdot n^2$	konstant
b. Shellsort	$O(n^{\frac{3}{2}})$	$\leq \log(n)$
c. Quicksort (im Mittel)	$1{,}3863 \cdot n \cdot \log(n)$	2 log(n)
Mischen		
Verschmelzen (merge sort)	$O(n \cdot \log(n))$	n
Streuen und Sammeln	O(n)	O(n)

3.4.2 Sortieren durch Aussuchen/Auswählen

Vorgehen:
Wähle das kleinste Element aus, stelle es an die erste Stelle und mache genauso mit den restlichen Elementen weiter.

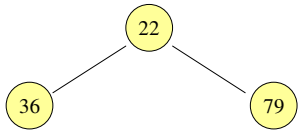
3.4.2.1 Sortieren durch "Minimum sortieren":

```
for i in 1..n-1 loop
    min := A(i); pos := i;          -- finde das kleinste Element von A(i) bis A(n)
    for j in i+1..n loop
        if A(j) < min then min:=A(j); pos := j; end if;
    end loop;                       -- das kleinste Element steht an Position pos
    A(pos) := A(i); A(i) := min;    -- nun steht das kleinste Element an Position i
end loop;
```

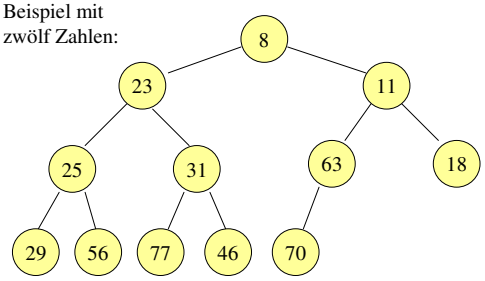
Zahl der Vergleiche stets $\frac{1}{2} \cdot n \cdot (n-1)$ Schritte: $\Theta(n^2)$
Platzaufwand 4 zusätzliche Speicherplätze: $O(1)$

Überlegung:
Könnte man die Information "Minimum sein" besser anordnen?

Ja, in einem binären Baum. Betrachte einen Konten mit zwei Nachfolgern. Schreibe in den Vaterknoten das Minimum der drei Knoten.



Beispiel mit zwölf Zahlen:



Als Feld levelweise aufgeschrieben:

8	23	11	25	31	63	18	29	56	77	46	70
1	2	3	4	5	6	7	8	9	10	11	12

Besonderheit dieses Baums: Auf jedem Pfad von der Wurzel zu einem Blatt sind die Elemente aufsteigend geordnet.

8	23	11	25	31	63	18	29	56	77	46	70
1	2	3	4	5	6	7	8	9	10	11	12

Die Bedingung "Der Inhalt eines Knotens ist stets kleiner als der Inhalt jedes Nachfolgeknotens" lässt sich präzisieren durch $A(i) \le A(2i)$ und $A(i) \le A(2i+1)$. Folgen oder Felder mit dieser Eigenschaft nennen wir "Heap" (meist übersetzt mit "Haufen"; sie haben nichts mit der Halde aus Kapitel 3.1 zu tun, die im Englischen ebenfalls "heap" heißt).

Definition 3.4.2.2:

Eine Folge oder ein array A(1), A(2), A(3), ..., A(n) heißt ein (aufsteigender) **Heap**, wenn für jedes i gilt:

$A(i) \le A(2i)$ und $A(i) \le A(2i+1)$,

wobei natürlich nur solche Ungleichungen betrachtet werden, bei denen 2i bzw. 2i+1 nicht größer als n sind.

Eine Folge oder ein array A(1), A(2), A(3), ..., A(n) heißt ein absteigender **Heap**, wenn für jedes i gilt:

$A(i) \ge A(2i)$ und $A(i) \ge A(2i+1)$,

wobei natürlich nur solche Ungleichungen betrachtet werden, bei denen 2i bzw. 2i+1 nicht größer als n sind.

3.4.2.3 Heapsort:

Gegeben sei ein Feld A(1), A(2), A(3), ..., A(n) mit Elementen aus einer geordneten Menge.

1. Wandle dieses Feld in einen absteigenden Heap um, so dass anschließend gilt: $A(i) \ge A(2i)$ und $A(i) \ge A(2i+1)$ für alle i (sofern $2i \le n$ bzw. $2i+1 \le n$ ist).
2. Für j von n abwärts bis 2 wiederhole:
   Vertausche A(1) und A(j).
   (Nun verletzt A(1) in der Regel die Heapeigenschaft.)
   Wandle das Feld A(1..j-1) ausgehend von der Wurzel so um, dass wieder ein absteigender Heap entsteht.

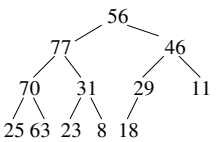
Im Folgenden beschreiben wir das Umwandeln in einen Heap (1.) und die Wiederherstellung der Heap-Eigenschaft (2.).

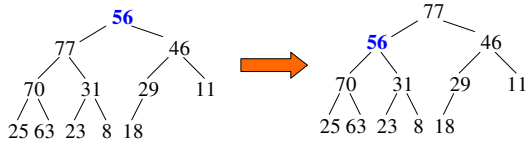
Die zentrale Prozedur ist die Herstellung der Heap-Eigenschaft in dem Teil des Feldes A, das mit dem Index "links" beginnt und mit dem Index "rechts" endet, unter der Annahme, dass höchstens beim Index "links" die Heap-Eigenschaft verletzt ist.

Hierfür vergleiche man den Inhalt des Elements A(links) mit den Inhalten der beiden Nachfolgeknoten und lässt gegebenenfalls den Inhalt von A(links) durch Vertauschen mit dem kleineren der beiden Nachfolger-Inhalten "absinken".

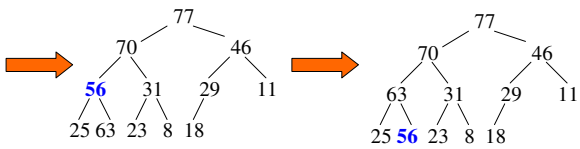
Betrachte ein Beispiel:

Die Heap-Eigenschaft ist nur bei 56 (links = 1 und rechts = 12) verletzt.





Vorgehen: Vergleiche 56 mit 77 und 46. Das Maximum ist 77, daher werden 77 und 56 vertauscht und mit 56 weitergemacht.



3.4.2.4: Prozedur für das Absinken

```
procedure sink (links, rechts: 1..n) is          -- A und n sind global
i, j: natural;  weiter: Boolean:=true;  v: <Elementtyp>;
begin  v := A(links);  i := links;  j := i+i;
   while  (j <= rechts) and weiter  loop
      if  j = rechts  then
         if A(j) > v then A(i):=A(j); i:=j; end if;
         weiter:=false;
      elsif A(j) < A(j+1) then
            if v < A(j+1) then  A(i) := A(j+1); i := j+1;
            else weiter:=false; end if;
      else   if v < A(j) then A(i) := A(j); i := j;
            else weiter:=false; end if;
      end if;
      j := i+i;                  -- v wird erst am Ende explizit eingetragen.
   end loop;                     -- i gibt am Ende die aktuelle Position an, an
   A(i):=v;                      -- der v einzufügen ist.
end sink;                        -- Im Inneren der Schleife werden bis zu zwei
                                 --Vergleiche zwischen Elementen durchgeführt.
```


3.4.2.5: Prozedur für Heapsort

```
procedure heapsort is                            -- A und n sind global
  procedure sink ... begin .... end sink;        -- siehe oben 3.4.2.4
  h: natural;  x: <Elementtyp>;
  begin
     h := n div 2;                               -- baue einen Heap auf
     for k in reverse 1..h loop  sink(k, n); end loop;
     for k in reverse 2..n loop
        x := A(1); A(1) := A(k); A(k) := x;      -- vertausche A(1) und A(k)
        sink(1, k-1);  end loop;
end heapsort;
```

Hinweis: Es lassen sich noch einige Umspeicherungen vermeiden, z.B. indem man das Wechselspiel zwischen v und x optimiert. Dies ändert aber nichts an der Zahl der (Element-) Vergleiche.

Wie viele Vergleiche benötigt Heapsort?

1. Aufbau des Heaps (for k in reverse 1..h loop sink (k, n); end loop;)

   Für k von h bis h/2: maximal 2 Vergleiche
   für k von h/2 bis h/4: maximal 4 Vergleiche
   für k von h/4 bis h/8: maximal 6 Vergleiche ....
   für k von $h/2^{i-1}$ bis $h/2^i$ maximal 2i Vergleiche (i=1, 2, ..., log(n))

Aufsummieren ergibt maximal 2·n Vergleiche: (beachte h = n/2)
$2\cdot h/2 + 4\cdot h/4 + 6\cdot h/8 + 8\cdot h/16 + ... + 2\cdot\log(n)\cdot 1$
$= 2h\cdot(2 - 2\cdot(\log(n)+1)/n) = 2\cdot n - 2\cdot\log(n) - 2 \le 2\cdot n$ Vergleiche.
Der Aufbau des Heaps erfolgt also in linearer Zeit.

Dies beweist man genauso wie die entsprechende Formel $\sum_{j=1}^{k} j\cdot 2^{j-1}$ in Abschnitt 1.3.2.3.d. Es gilt:
$1/2^1 + 2/2^2 + 3/2^3 + 4/2^4 + ... + m/2^m = 2 - (m+1)/2^{(m-1)}$.

2. Sortierphase (for k in reverse 2..n loop ... sink(1, k-1) end loop;)

   Für k von n bis n/2: maximal 2·log(n) Vergleich
   für k von n/2 bis n/4: maximal 2·log(n)-1 Vergleiche
   für k von n/4 bis n/8: maximal 2·log(n)-2 Vergleiche ....
   für k von $n/2^{i-1}$ bis $n/2^i$ maximal 2·i Vergleiche (i=1, 2, ..., log(n))

Aufsummieren ergibt maximal 2·n·log(n) Vergleiche: Sei n > 1:

$\log(n)\cdot n + (\log(n)-1)\cdot n/2 + (\log(n)-2)\cdot n/4 + (\log(n)-3)\cdot n/8 + ... + 2 =$
$2\cdot n\cdot(\log(n)/2 + \log(n)/4 + ... + 1/2^{\log(n)} - 1/4 - 2/8 - 3/16 - ... - (\log(n)-1)/2^{\log(n)} =$
$2\cdot n\cdot\log(n)\cdot(1-1/2^{\log(n)}) - n\cdot(1/2^1 + 2/2^2 + 3/2^3 + 4/2^4 + ... + (\log(n)-1)/2^{\log(n)-1}) =$
$2\cdot n\cdot\log(n) - 2\cdot\log(n) - n\cdot(2 - \log(n)/2^{\log(n)-2}) = 2\cdot n\cdot\log(n) - 2\cdot n + 2\cdot\log(n)$
$< 2\cdot n\cdot\log(n)$

(Wir benutzten hier erneut die Formel von der vorherigen Folie ganz unten.)

Insgesamt ergeben sich für Heapsort im worst case maximal
$(2\cdot n - 2\cdot\log(n) - 2) + (2\cdot n\cdot\log(n) - 2\cdot n + 2\cdot\log(n)) =$
$2\cdot n\cdot\log(n) - 2$ Vergleiche (für n>1).
Im best case kann n·log(n) erreicht werden, da durch spezielle Folgen das Absinken beschränkt werden kann. Der Mittelwert allerdings wird ebenfalls bei 2·n·log(n) liegen, da Heapsort keine Vorsortierungen oder günstigen Konstellationen ausnutzt. Dies deckt sich auch mit Experimenten. Somit erhalten wir den

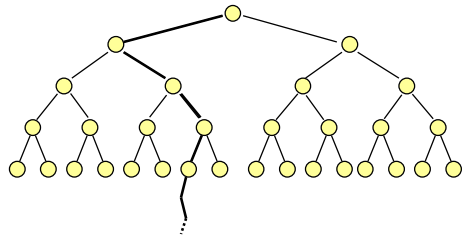
Satz 3.4.2.6:
Dieses normale Heapsort (aus dem Jahre 1962) benötigt im schlechtesten Fall höchstens 2·n·log(n) Vergleiche (für n>1).
(Im besten Fall kann man höchstens den Faktor 2 sparen. Der average case liegt recht nahe beim schlechtesten Fall.)

Geht es nicht doch noch besser?

Ja, man kann den Faktor 2 noch verkleinern (allerdings kann er nicht kleiner als "1" werden, wie in Satz 3.4.1.11 gezeigt wurde.)

Beobachtung: Beim eigentlichen Sortieren wird das letzte Element mit dem ersten vertauscht. Die Prozedur "sink" wird dieses letzte Element in der Regel sehr weit absenken, da es ja zu den leichten Elementen gehört hat, die sich ganz unten im Baum befinden. Man sollte daher das Element nicht von oben nach unten absenken, sondern es von unten nach oben (also "bottom-up") aufsteigen lassen. Hierzu muss man aber die Stelle, an der es einzufügen ist, kennen. Genau diese Stelle ermittelt man durch die Berechnung des "Einsinkpfads".

3.4.2.7 Einsinkpfad: Dies ist der Weg, den ein Element, das in die Wurzel gesetzt wurde, nehmen muss, damit die Heap-Eigenschaft wiederhergestellt wird (vgl. 3.4.2.4).
Dieser Pfad ist unabhängig vom einzusortierenden Element. Er endet in einem Blatt des Baumes. Es genügt, den Index dieses Blattes zu bestimmen.



Ermittlung des Einsinkpfads:

starte mit der Wurzel;
while noch nicht Blatt erreicht loop
    vergleiche die Inhalte der beiden Nachfolgeknoten;
    nimm den Knoten mit dem größeren Inhalt;
end loop;

Bei der Darstellung mit Feldern braucht man nur den Index j des letzten Elements des Einsinkpfads zu kennen. Die anderen Knoten auf diesem Pfad besitzen die Indizes
j div 2, (j div 2) div 2, ((j div 2) div 2) div 2, ..., 1.

Die folgende Funktion berechnet den Index j des letzten Elements des Einsinkpfads.

Ermittlung des Einsinkpfads:

```
function einsinkpfad (rechts: 1..n) return 1..n is
j: 1..n := 1; m: natural := 2;
begin
    while  m < rechts  loop
        if  A(m) < A(m+1)  then  j := m+1;  else  j:= m;  end if;
        m := j+j;
    end loop;
    if  m = rechts  then  j := rechts;  end if;
    return j;
end;
```


3.4.2.8 Bottom-up-Heapsort: (Carlsson 1987, Wegener, 1993)

1. Ermittle den Index j des letzten Elements des Einsinkpfads.
2. Suche von j aus entlang des Einsinkpfads die Stelle, wo das einzusortierende Element hingehört.
3. Füge es dort ein und schiebe alle darüber stehenden Elemente entlang des Einsinkpfads um eine Position in Richtung der Wurzel.

Für die Programmierung benutzen wir die obige Funktion "einsinkpfad", die wir jedoch direkt in den Algorithmus integrieren. Weiterhin führen wir die Verschiebung von Punkt 3. bereits beim Berechnen von j durch, da man in der Regel den Einsinkpfad nur wenige Schritte zurücklaufen muss und hierdurch im Mittel ein doppeltes Durchlaufen vermieden wird.


```
procedure bottumupheapsort is                -- A und n sind global
procedure sink ... begin .... end sink;      -- wie früher
h, j, m: natural; x: <Elementtyp>;
begin  h := n div 2;                         -- baue einen Heap auf
    for k in reverse 1..h loop  sink (k, n); end loop;
    for k in reverse 2..n loop               -- x = A(k) einsinken lassen
        x := A(k); A(k) := A(1);             -- rette A(1) nach A(k)
        j := 1; m:= 2;                       -- Suche den Index j
        while  m < k-1  loop
            if  A(m) < A(m+1)  then  A(j) := A(m+1); j := m+1;
            else  A(j) := A(m); j:= m;  end if;
            m := j+j;
        end loop;
        if  m = k-1  then  A(j) := A(k-1); j := k-1;  end if;
-- Nun ist der Index j (=Ende des Einsinkpfads) bekannt und alle Inhalte auf dem
-- Einsinkpfad sind um eine Position in Richtung der Wurzel verschoben worden.
```



```
-- Die Elemente des Einsinkpfads müssen nun zurückgeschoben werden,
-- solange die Stelle, an die x gehört, noch nicht erreicht ist.
        while  (j > 1) and then (A(j) < x)  loop
            i := j div 2; A(j) := A(i); j := i;
        end loop;
-- Die Stelle j, an die x gehört, ist nun erreicht.
    A(j) := x;
    end loop k;
end bottomupheapsort;
```

Wie viele Vergleiche benötigt Bottom-up-Heapsort?

1. Aufbau des Heaps: Genauso wie beim normalen Heapsort
maximal $2 \cdot n - 2 \cdot \log(n) - 2 \leq 2 \cdot n$ Vergleiche.

2. Sortierphase

Hier benötigt man maximal für jedes k so viele Vergleiche, wie die doppelte Länge des Einsinkpfads ist, also rund $2 \cdot \log(k)$.
Dies führt zunächst nur auf genau die gleiche Abschätzung wie beim normalen Heapsort.
Aber: In den meisten Fällen wird man bereits nach etwas mehr als log(k) Vergleichen fertig sein.
Experimente bestätigen, dass der Faktor "2" vom normalen Heapsort im Mittel auf "1" sinkt; dies lässt sich auch beweisen.
Es gibt aber Folgen, bei denen man nur auf den Faktor 1,5 kommen kann.

Man kann beweisen: Bottom-up-Heapsort benötigt im worst case maximal $1{,}5 \cdot n \cdot \log(n)$ Vergleiche. Dies tritt aber fast nie auf, so dass man im Mittel von $n \cdot \log(n) + O(n)$ Vergleichen ausgehen kann. Carlsson konnte $n \cdot \log(n) + 0.67n$ als obere Schranke im average case nachweisen.

3.4.2.9 Satz:
Bottom-up-Heapsort benötigt
im schlimmsten Fall höchstens $1{,}5 \cdot n \cdot \log(n)$,
im Mittel höchstens $n \cdot \log(n) + 0.67 \cdot n$ Vergleiche.

Beachte: Heapsort und seine Varianten sind
garantierte $n \cdot \log(n)$ - Verfahren.

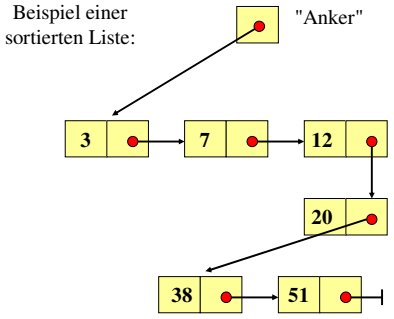
Hinweis: Es gibt weitere Varianten, z.B. von McDiarmid and Reed 1998 oder von Katajainen 1998. Ziel ist es, die untere theoretische Schranke von Satz 3.4.1.11 zu erreichen. Der Faktor 1 (statt 1,5) wurde mit dem "ultimativen" Heapsort erreicht, das aber für die Praxis bisher ungeeignet ist.

3.4.3 Sortieren durch Einfügen

Wenn die Elemente einer Folge durch Zeiger (sortierte Listen, Suchbäume) dargestellt werden, so wird man die einzelnen Elemente der Folge nacheinander in diese Struktur einfügen und dabei die Struktur stets wiederherstellen.

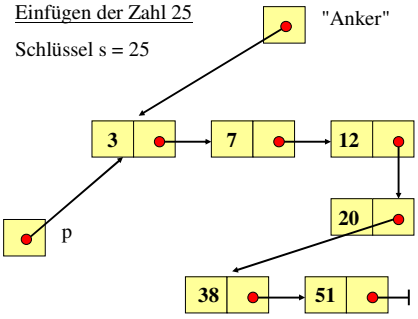
Einfachster Fall: Einfügen in eine sortierte Liste.
Eine Liste heißt sortiert, wenn für alle Elemente der Liste gilt: p.Inhalt ≤ p.next.Inhalt. Hiebei ist p ein Zeiger, der auf das jeweilige Element der Liste verweist.

Erinnerung: Datenstruktur für eine Liste, vgl. 1.3.3.1:
type Zelle;
type Ref_Zelle is access Zelle;
type Zelle is record Inhalt: Integer; Next: Ref_Zelle; end record;



Einfügen der Zahl 25

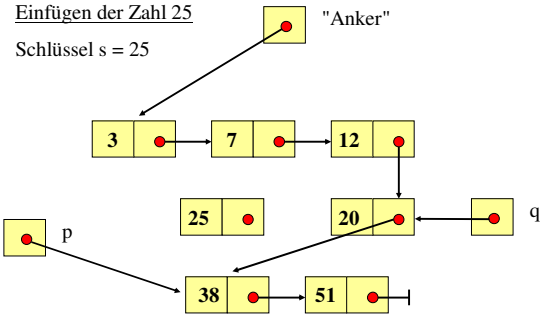
Schlüssel s = 25



p := Anker;
while p /= null and then s > p.Inhalt loop p:=p.Next; end loop;

Einfügen der Zahl 25

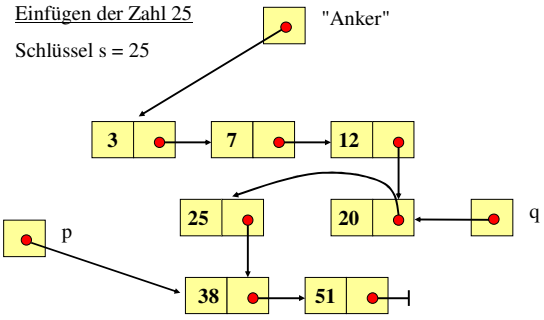
Schlüssel s = 25



p := Anker;
while p /= null and then s > p.Inhalt loop p:=p.Next; end loop;

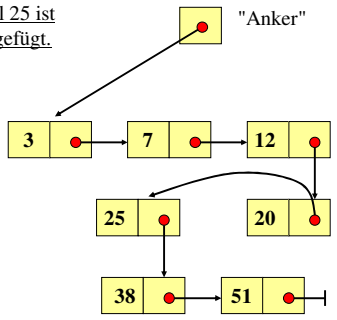
Einfügen der Zahl 25

Schlüssel s = 25




```
p := Anker;
while p /= null and then s > p.Inhalt loop p:=p.Next; end loop;
```


Die Zahl 25 ist nun eingefügt.



Prozedur zum Einfügen eines *Elementes in eine sortierte Liste:*

```
procedure Einf (Anker: in out Ref_Zelle; s: in Integer) is
p, q: Ref_Zelle;
begin   p := Anker; q := null;
        while p /= null and then s > p.Inhalt loop
                        q := p;  p := p.Next;  end loop;
        if  q = null  then  Anker := new Zelle'(s, null);
        else  q.Next := new Zelle'(s, p);  end if;
end Einf;
```

3.4.3.1 Sortieren von n Elementen durch Einfügen in eine Liste:

```
while not End_of_File loop Get(v); Einf(Anker, v); end loop;
```

Zahl der Vergleiche im Mittel und im schlechtesten Fall: $O(n^2)$.

3.4.3.2 Sortieren durch Einfügen in einen Baum: siehe 1.6.4.4.

Wenn man beliebige binäre Suchbäume verwendet, so können diese im schlechtesten Fall zu einer Liste entarten und daher beträgt im worst case die Zeitkomplexität $O(n^2)$.

Benutzt man aber anstelle eines beliebigen Suchbaums AVL-Bäume, so ist deren Höhe durch $1.4404 \cdot n \cdot \log(n)$ nach Satz 3.2.4.7 beschränkt. Daher ist die Anzahl der Vergleiche für das Sortieren mit Bäumen auch im schlechtesten Fall durch $1.4404 \cdot n \cdot \log(n) + O(n)$ beschränkt.

In der Praxis kann man bei der Verwendung von beliebigen Suchbäumen im Mittel mit $1.3863 \cdot n \cdot \log(n) + O(n)$, bei der Verwendung von AVL-Bäumen im Mittel mit $n \cdot \log(n) + O(n)$ rechnen (siehe Satz 3.2.2.15 und Hinweis nach Satz 3.2.4.7).

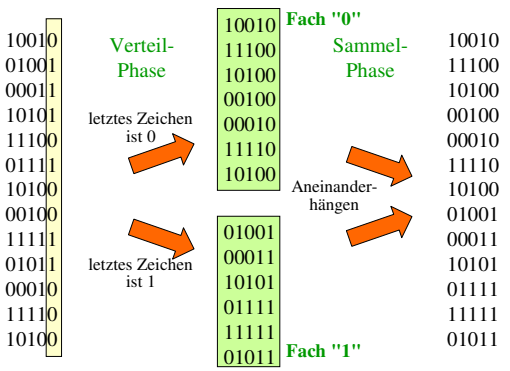
3.4.3.3 Sortieren durch Fachverteilen

Sind die Schlüssel Wörter über einem endlichen Alphabet $A = \{a_1, a_2, ..., a_s\}$, kann man die zu sortierenden Elemente zunächst bzgl. des letzten Zeichens an s Listen anfügen. Diese Listen hängt man aneineinader und fügt nun alle Elemente bzgl. des vorletzten Zeichens an s Listen an usw. Liegt die Länge jedes Schlüssels in der Größenordnung von $\log(n)$, dann ergibt sich ein $O(n \cdot \log(n))$-Sortierverfahren.

Das Anhängen an die s Listen entspricht dem Ablegen in s verschiedene Fächer, weshalb dieses Verfahren als "Fach-verteilen" bezeichnet wird.

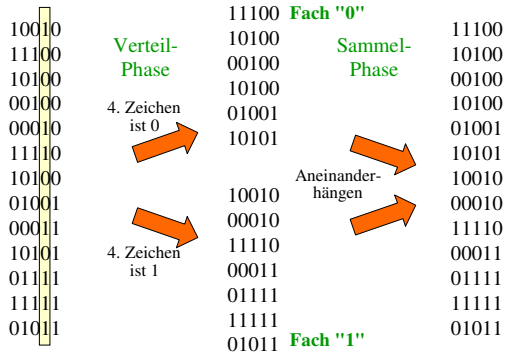
Wir erläutern das Verfahren nur an einem Beispiel mit s=2. Die Programmierung ist nicht schwierig.

Gegeben seien 13 Schlüssel als 0-1-Darstellungen der Länge 5



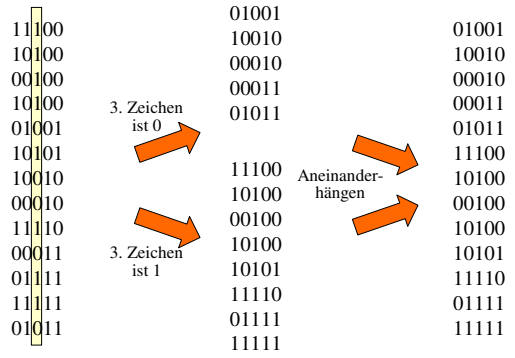
Nun iterativ weiter durch alle Stellen der Schlüssel.

Gegeben seien 13 Schlüssel als 0-1-Darstellungen der Länge 5

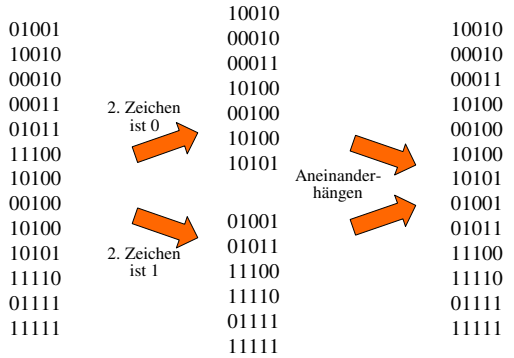


Nun iterativ weiter durch alle Stellen der Schlüssel.

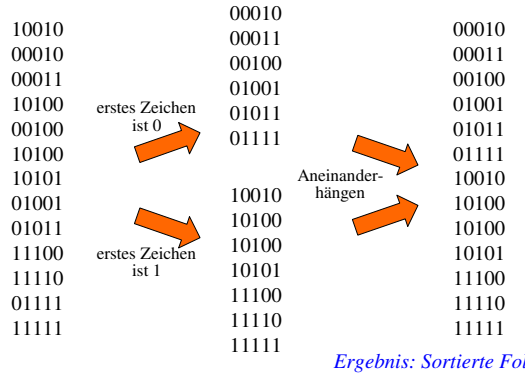
Gegeben seien 13 Schlüssel als 0-1-Darstellungen der Länge 5



Gegeben seien 13 Schlüssel als 0-1-Darstellungen der Länge 5



Gegeben seien 13 Schlüssel als 0-1-Darstellungen der Länge 5



Ergebnis: Sortierte Folge.

Man beachte: Die Sortierung jeder Phase muss stabil sein. Hat man s Zeichen (z.B. s=26 für das Alphabet oder s=128 für den ASCII-Zeichensatz), dann muss man s solche "Fächer" bereitstellen. In der Regel organisiert man die Fächer als Listen, an die man hinten den jeweils nächsten gelesenen Schlüssel anhängt; danach werden die s Listen aneinandergehängt und die nächste Verteilphase kann beginnen.
Das Sortieren erfordert $s \cdot n$ Vergleiche. Es eignet sich besonders gut für das Sortieren von binärer Information (z.B. in der Systemprogrammierung) und in allen Fällen, in denen $s \leq \log(n)$ ist. Nachteilig ist, dass man in der Regel das Doppelte an Speicherplatz benötigt. Man kann aber auch einen Austausch wie bei Quicksort vornehmen, so dass die 0-en oben und die 1-en unten zu stehen kommen; im Falle s>2 bietet sich auch ein bucket-sort an, siehe 3.4.6.

3.4.4 Sortieren durch Austauschen

Die beiden wichtigsten Vertreter Bubble Sort und Quicksort wurden bereits vorgestellt; sie sind auf den nächsten Folien nochmals als Programm ausformuliert.

Aufwandsabschätzung für Bubble Sort:
Platzbedarf: konstant (3 zusätzliche Variablen)
Zeitbedarf: worst case: $\frac{1}{2} \cdot n \cdot (n-1)$ Vergleiche,
best case: n, falls das Feld bereits sortiert ist.
im Mittel sind $\frac{1}{4} \cdot n \cdot (n-1)$ Vergleiche zu erwarten.

Man könnte das Feld abwechselnd aufwärts und abwärts durchlaufen (dies nennt man Shaker Sort). Diese Variante bringt aber faktisch keine Verbesserung des Laufzeitverhaltens.

3.4.4.1 Bubble Sort für ein Integer-Feld A mit dem Indextyp 1..n (das Feld A sei vom Feld-Typ Vektor):

```
procedure BubbleSort (A: in out Vektor) is
Weiter: Boolean := True;  H: Integer;
begin
   while Weiter loop
      Weiter := False;
      for i in 1..n-1 loop
         if A(i) > A(i+1) then Weiter := True;
            H := A(i); A(i) := A(i+1); A(i+1) := H;
         end if;
      end loop;
   end loop;
end BubbleSort;
```

Oft programmiert man Buble Sort auch "abwärts", also:
for i in revers 1..n-1 loop ...

3.4.4.2 Quicksort: Aus 1.7.3.4 übernehmen wir mit den Datentypen *type Index is 1..n; ... A: array (Index) of Integer;* ... das Programm:

```
procedure Quicksort (L, R: Index) is     -- A ist global, A(L..R) wird sortiert
i, j: Index; p, h: Integer;              -- p wird das Pivot-Element
begin
  if L < R then
     i := L; j := R; p := A((L+R)/2);    -- man kann p auch anders wählen
     while i <= j loop                   -- die Indizes i und j laufen aufeinander zu
        while A(i) < p loop i := i+1; end loop;
        while A(j) > p loop j := j-1; end loop;
        if i <= j then h:=A(i); A(i):=A(j); A(j):=h;
                       i := i+1; j := j-1;  end if;
     end loop;                           -- auch bei Gleichheit A(i)=p oder A(j)=p vertauschen!
     if (j-L) < (R-i) then Quicksort(L, j); Quicksort(i, R);
     else Quicksort(i, R); Quicksort(L, j); end if;  -- Vorsicht; siehe unten!
  end if;
end Quicksort;
... Quicksort(1,n); ...                  -- Aufruf des Sortierverfahrens
```


3.4.4.3 Platzbedarf von Quicksort

Platzbedarf: maximal 2·log(n) laut Hinweis 4 in 1.7.3.3.
Dies muss hier aber noch einmal genau durchdacht werden. In 1.7.3.3 wurde suggeriert, dass mit dem rekursiven Aufruf

Quicksort(L, j); Quicksort(i, R);

die Paare (L, j) und (i, R) auf einen Stack gelegt werden, dann das oberste Paar entfernt und weiter gearbeitet wird, so dass das zuvor dort abgelegte Paar (L, R) also durch (L, j) und (i, R) ersetzt würde. Wegen der Abfrage "if (j-L) < (R-i) then" kämen nur höchstens logarithmisch viele Paare übereinander zu liegen, d.h., der Stack für die Rekursion wäre durch 2·log(n) Plätze beschränkt. Dies trifft aber nur zu, wenn das Paar (L, R) bei dem rekursiven Aufruf entfernt würde. Genau dies geschieht aber bei obigem Programm nicht, weil das "end" der Prozedur noch nicht erreicht ist; vielmehr bleiben (i, R) und (L, R) übereinander liegen und im worst case braucht man weiterhin n Speicherplätze.

Platzbedarf von Quicksort (Fortsetzung):

Man muss also das Quicksort-Programm so abändern, dass man den Stack für die Rekursion im wesentlichen selbst verwaltet und nutzlos gewordene Information nicht in den Stack schreibt bzw. in ihm stehen lässt. Es ist klar, dass man auf diese Weise die Tiefe des Stacks auf log(n) Paare beschränken kann, so dass die in 1.7.3.3 gemachte Aussage korrekt bleibt, allerdings nicht mit dem dort bzw. in 3.4.4.2 angegebenen Programm.

Aufgabe: Versuchen Sie, eine Lösung für dieses Problem anzugeben, d.h., Sie sollen die Prozedur Quicksort so abwandeln, dass die rekursive Tiefe des Aufruf-Stacks durch log(n) beschränkt bleibt.
(Hinweis: Eine Lösung finden Sie im Buch Ottmann/Widmaier.)

3.4.4.4 Zeitbedarf von Quicksort:

Zeitbedarf: worst case: ≈ ½·n² Vergleiche, wenn das Pivot-Element stets das kleinste oder das größte Element des Teilfelds ist.
best case: n·log(n), wenn das Pivot-Element stets das mittelste der Elemente des Teilfelds ist.
average case: Es ist mit 1.3863·n·log(n) - 1,8456·n Vergleichen im Mittel zu rechnen. Begründung:

Man kann das Aufteilen des Feldes in zwei Teilfelder mit dem Aufbau eines binären Suchbaums vergleichen, wobei das Pivot-Element in die Wurzel kommt und aus den beiden Teilfeldern rekursiv der linke und rechte Unterbaum aufgebaut werden. Quicksort verhält sich daher im Mittel genau wie das Baumsortieren mit binären Suchbäumen (Satz 3.2.2.15). Die formalen Berechnungen liefern das gleiche Ergebnis, siehe Lehrbücher.

Zeitbedarf von Quicksort im Mittel (Fortsetzung):

Für den Zeitbedarf ist die "gute Wahl" des Pivot-Elements p entscheidend. Gebräuchlich ist folgende Variante: Statt das Element p irgendwie fest zu wählen (z.B.: p:=A((L+R)/2) oder p:=A(L) oder ...) nimmt man das mittlere von drei Elementen, z.B. von A(L), A(R) und A((L+R)/2).
Mit dieser "Median aus 3"-Variante erhält man einen deutlich besseren mittleren Zeitbedarf, nämlich im Mittel höchstens
1.188·n·log(n) - 2,255·n Vergleiche.
Diese Variante wird daher gern in der Praxis verwendet.

Aufgabe: Erweitern Sie unsere Prozedur um diese Variante. Machen Sie Messungen mit zufällig erzeugten Daten und verifizieren Sie hiermit den Laufzeitgewinn, der ab einem gewissen n eintritt. (Bestimmen Sie dieses n experimentell.)

Zeitbedarf von Quicksort im Mittel (Fortsetzung):

Eine andere Variante besteht darin, die Zerlegung in drei Teilfelder vorzunehmen (sog. "Dreiwege-Split"):
Alle Elemente in A(L..j) sind echt kleiner als p,
alle Elemente in A(i..R) sind echt größer als p und
alle Elemente in A(j+1..i-1) sind gleich p, wobei dieser Bereich ja nie leer ist. Da man ihn nicht bei der Rekursion berücksichtigen muss, verringert sich die Rekursionstiefe und damit die Laufzeit. Treten in der Folge viele Schlüssel mehrfach auf, so lässt sich hierdurch das Sortieren deutlich beschleunigen.

Aufgabe: Erweitern Sie unsere Prozedur auch um diese Variante. Die Schwierigkeit liegt darin, alle Schlüssel, die gleich p sind, ohne Zusatzaufwand in der Mitte des zu sortierenden Teilfelds zu platzieren. Auch hier sollten dann Sie Messungen mit zufällig erzeugten Daten vornehmen und prüfen, ab welchem Prozentsatz gleicher Schlüssel sich dieses Vorgehen lohnt.

Hinweise:

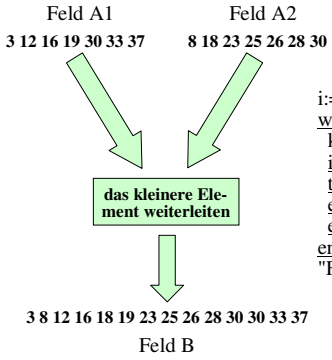
Historisch gesehen gibt es weitere Sortierverfahren, die auf dem Austauschen beruhen. Am bekanntesten ist Shellsort, bei dem die Folge in äquidistante Teilfolgen unterteilt wird und als erste Phase in diesen eine Sortierung erfolgt (z.B. ein Bubble Sort Durchlauf). Die äquidistanten Abstände werden dann für eine zweiten Phase verringert und diese Verringerung wird fortgesetzt, bis der Abstand gleich 1 ist, d.h. eine Sortierung der bis dahin entstandenen schon gut vorsortierten Folge stattfindet. Dieses Vorgehen setzt in den einzelnen Phasen ordnungsverträgliche Sortierverfahren voraus.
Denken Sie sich selbst "schnelle" heuristische Verfahren aus. Der Fantasie sind hier kaum Grenzen gesetzt. Führen Sie aber auf jeden Fall Messungen durch (die meist die erhofften Beschleunigungen nicht bestätigen).

3.4.5 Mischen

Die bisherigen Sortierverfahren haben einzelne Elemente verglichen, die oft weit voneinander entfernt in der zu sortierenden Folge standen. Sie sind daher für riesige Datenbestände, die auf Hintergrundspeichern stehen, nur bedingt einsetzbar.

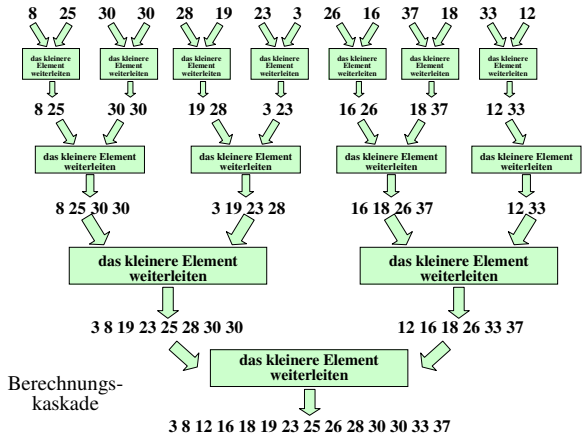
Daten werden von Hintergrundspeichern "stromartig" eingelesen, vergleichbar dem Lesen von Magnetbändern. Deshalb muss man immer möglichst viele Daten, die zusammenhängend verglichen und sortiert werden können, zusammenfassen und verarbeiten.

Hierfür ist vor allem das Zusammen-Mischen zweier bereits sortierter Datenströme geeignet.




```
i:= 1; j := 1; k := 0;
while i <=n and j <=n loop
  k := k+1;
  if A1(i) < A2(j)
  then B(k) := A1(i); i := i+1;
  else  B(k) := A2(j); j := j+1;
  end if;
end loop;
"Füge den Rest an B an"
```

Der gesamte Sortierprozess ist auf der nächsten Folie dargestellt.



3.4.5.1: Verschmelzen zweier Folgen vom Typ Vektor

```
procedure Verschmelzen (A1, A2: in Vektor; B: in out Vektor;
                        LA1, RA1, LA2, RA2, LB: in Integer;
                        RB: out Integer) is
i, j, k: Integer;
begin i:= LA1; j := LA2; k := LB-1;
  while i <=RA1 and j <=RA2 loop
    k := k+1;
    if A1(i) < A2(j) then B(k) := A1(i); i := i+1;
                     else  B(k) := A2(j); j := j+1; end if;
  end loop;                  -- nun muss der Rest einer Folge noch angefügt werden
  if i <= RA1  then
       for m in i..RA1 loop k:=k+1; B(k):=A1(m); end loop;
  else  for m in j..RA2 loop k:=k+1; B(k):=A2(m); end loop; end if;
  RB := k;                   -- RB dient nur für Kontrollzwecke, kann entfallen
end Verschmelzen;
```

3.4.5.2: Sortieren durch Mischen

Es soll ein Feld A(1..n) durch Mischen sortiert werden.
Zuerst die "*Bottom-Up*-Denkweise", die zu einem Iterationsverfahren führt: Man verschmilzt zunächst je zwei Folgen der Länge 1 zur sortierten Folgen der Länge 2 (man muss hierfür n/2 mal "Verschmelzen" aufrufen),
dann verschmilzt man je zwei sortierte Folgen der Länge 2 zu sortierten Folgen der Länge 4 (n/4 mal "Verschmelzen" aufrufen),
danach das Gleiche für sortierte Folgen der Länge 4, 8, 16 usw., bis zwei Folgen der Läng n/2 zu einer sortierten Folge verschmolzen wurden.
Sofern n eine Zweierpotenz war, funktioniert dieses Verfahren bereits; im allgemeinen Fall muss man beim Verschmelzen jeweils Folgen, deren Länge sich bis zum Faktor 2 unterscheiden darf, berücksichtigen (vgl. Beispiel).
Dieses Mischen heißt in der Literatur "straight mergesort".

Sortieren durch Mischen (Fortsetzung)

Nun die "*Top-Down*-Denkweise": Um ein Feld zu sortieren, sortiert man zuerst die linke Hälfte, dann die rechte Hälfte und verschmilzt die beiden sortierten Folgen. Das Ergebnis ist eine rekursive Prozedur.
Da dieses Vorgehen einfacher zu verstehen und aufzuschreiben ist, wird die rekursive Version im folgenden realisiert.
Der Ansatz ist einfach. Die Hauptschwierigkeit besteht hier in der präzisen Angabe der jeweiligen Teilfeld-Grenzen besteht.
Wir verzichten auf volle Allgemeinheit und wollen "nur" das Feld A(L..R) sortieren. Sortierte Teilfelder A(x..y) und A(u..v) mischen wir in ein Hilfsfeld B(L..R) und schreiben danach das Ergebnis wieder nach A zurück.

3.4.5.3: Programm zum Sortieren durch Mischen ("Mergesort")

```
procedure Mergesort (L, R: in Integer) is
Mitte: Integer; i, j, k: Integer;          -- Die Felder A und B sind global.
begin                                      -- Es wird A(L..R) sortiert.
if R > L then  Mitte := (L+R)/2;           -- Sortiere rekursiv zwei Hälften der Folge
   Mergesort(L, Mitte);  Mergesort(Mitte+1, R);
   i := L; j := Mitte+1; k := L-1;         -- Nun mischen von A nach B, wie in 3.4.5.1
   while i <= Mitte and j <= R loop  k := k+1;
      if A(i) < A(j) then B(k) := A(i); i := i+1;
                     else B(k) := A(j); j := j+1; end if;
   end loop;
   if i <= Mitte then
      for m in i..Mitte loop k:=k+1; B(k):=A(m); end loop;
   else for m in j..R loop k:=k+1; B(k):=A(m); end loop; end if;
   for m in L..R loop A(m) := B(m); end loop;    -- zurückkopieren nach A
end if;
end Mergesort;
```


Wir haben das Verfahren mit Feldern realisiert.

Es ist aber klar, dass man es auch leicht mit linearen Listen implementieren kann, wobei nur Zeiger umgesetzt werden müssen. Hierbei kann man alle Umspeicherungen vermeiden. Durchdenken Sie die Vor- und Nachteile eines solchen Verfahrens und entwerfen Sie ein Programm, für das die zu sortierenden Daten als einfach verkettete lineare Liste vorliegen.

3.4.5.4 Aufwandabschätzung für das Mischen:

Wenn V(n) die maximale Zahl der Vergleiche zum Sortieren von n Elementen ist, dann gilt:

$V(1) = 0$ und für alle $n > 1$: $V(n) = 2 \cdot V(n/2) + n - 1$.

Lösung dieser Gleichung: $V(n) = n \cdot \log(n) - n + 1$.

Mergesort ist also ein garantiertes $n \cdot \log(n)$-Verfahren.
Die Zahl seiner Vergleiche kommt sehr dicht an die untere theoretische Grenze heran, siehe Satz 3.4.1.11.

Die maximale Rekursionstiefe ist beim Mischen log(n), da in jedem Rekursionsschritt halbiert wird. Auch hieraus kann man ablesen, dass die maximale Zahl der Vergleiche durch $n \cdot \log(n) - n/2 - n/4 - n/8 - \ldots - 1 = n \cdot \log(n) - n + 1$ beschränkt ist.

Bei Mergesort sind viele Umspeicherungen erst beim Verschmelzen von A nach B und dann zurück nach A erforderlich. Wie hoch ist die Zahl der Speichervorgänge?

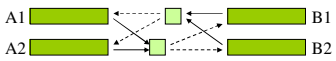
In jeder Rekursionstiefe werden alle Daten genau einmal nach B und wieder zurück transportiert. Folglich werden insgesamt stets $2 \cdot n \cdot \log(n)$ Umspeicherungen durchgeführt.

Dies scheint viel zu sein. Man mache sich aber klar, dass bei Quicksort je Rekursionstiefe bis zu n/2 Vertauschungen stattfinden, die jeweils 3 Umspeicherungen erfordern, weshalb Quicksort auch im günstigen Falle, dass die Rekursionstiefe durch log(n) beschränkt bleibt, bis zu $(3/2) \cdot n \cdot \log(n)$ Umspeicherungen ausführen kann.

3.4.5.5 Varianten des Sortierens durch Mischen

Das oben genannte Vorgehen bezeichnet man als "2-way-merge sort" (*Zwei-Wege-Mischen* oder auch Zwei-Phasen-Mischen), da eine Bewegung der Daten von Feld A nach B (erster Weg) und anschließend eine Bewegung der Daten zurück nach A (zweiter Weg) stattfindet.

Natürlich kann man die Rolle der Ziel-Felder in jedem Durchgang ändern, d.h.: Man verschmilzt zwei sortierte Folgen von A1 und A2 abwechselnd auf die Felder B1 und B2 und anschließend zwei sortierte Folgen von B1 und B2 zurück nach A1 bzw. A2 usw. So spart man die Hälfte der Umspeicheroperationen. Man muss sich hierbei die jeweiligen Grenzen merken und bis zu 2n zusätzliche Speicherplätze bereitstellen.



Die durchgezogenen Pfeile bezeichnen das Zusammenführen zweier sortierter Folgen. Das sortierte Ergebnis des Verschmelzens wird abwechselnd über einen der gestrichelten Pfeile weitergeleitet.

3-Wege-Mischen:

Man kann auch mit drei Feldern A, B und C arbeiten. Zuerst wird die zu sortierende Folge auf zwei Bänder A und B verteilt und zwar F_k sortierte Teilfolgen auf Band A und F_{k-1} sortierte Teilfolgen auf Band B (F_k ist die k-te Fibonaccizahl; falls die Anzahlen nicht "aufgehen", so fülle man mit "dummy"-Folgen auf die nächste Fibonaccizahl auf). Nun werden sortierte Teilfolgen von A und B nach C gemischt, bis das Feld B keine sortierte Folge mehr besitzt. Dann besitzen C genau F_{k-1} und A genau $F_k - F_{k-1} = F_{k-2}$ sortierte Teilfolgen. Nun werden die sortierten Teilfolgen von A und C nach B gemischt, bis das Feld A keine sortierte Teilfolge mehr besitzt (auf den Feldern B und C befinden sich nun F_{k-2} bzw. F_{k-3} sortierte Teilfolgen). Nun wird A das Ziel-Feld, d.h., es werden sortierte Teilfolgen von B und C nach A gemischt usw., bis am Ende auf einem der Bänder die sortierte Gesamtfolge steht.

Natürliches Mischen:

Eine Teilfolge $a_i\ a_{i+1} \ldots a_j$ der Folge $a_1\ a_2 \ldots a_n$ heißt ein Lauf, wenn dies eine maximal lange nicht fallende Teilfolge ist, d.h., wenn gilt: $a_{i-1} > a_i \le a_{i+1} \le \ldots \le a_j > a_{j+1}$ gilt (wenn i=1 oder j=n ist, so entfallen die entsprechenden Ungleichungen).

Statt Teilfolgen der Länge 1, dann der Länge 2 usw. zu mischen, kann man in jedem Durchgang die vorhandenen Läufe mischen, wodurch sich die Rekursionstiefe und damit die Zahl der Umspeicherungen verringert, die Zahl der Abfragen aber wegen des Tests, wo ein Lauf endet, insgesamt nicht geringer wird. Ein solches Ausnutzen der zufällig vorhandenen Teil-Sortierungen bezeichnet man als "natürliches Mischen". Dadurch wird das Mischen zu einem ordnungsverträglichen Sortierverfahren mit garantierter $n \cdot \log(n)$-Laufzeit.

Aufgabe: Programmieren Sie das "natürliche 3-Phasen-Mischen".

Parallelisieren:

Hat man einen Baustein, der das Verschmelzen zweier Folgen vornimmt, so kann man in jeder Rekursionstiefe alle Operationen parallel ausführen (siehe die Kaskade auf der Folie zu Beginn von 3.4.5).

Man benötigt dann n/2 solcher Bausteine zum Mischen von Teilfolgen der Länge 1, man braucht n/4 dieser n/2 Bausteine für das Verschmelzen aller Teilfolgen der Länge 2 usw. Durch geschicktes Zusammenschalten kann man also die Kaskade mit n/2 solcher Bausteine realisieren.

Wie lange dauert dann das Sortieren? Die erste Schicht benötigt genau 2 "Takte", die nächste 4, die nächste 8 usw. bis zur letzten Schicht mit n Takten. Folglich kann man wegen $2+4+8+\ldots+n/4+n/2+n = 2n-2$ die n Daten mit diesem Parallelverfahren in 2n-2 Schritten sortieren. Der "Preis" hierfür sind die n/2 Bausteine.

3.4.6 Streuen und Sammeln

Manchmal liegen die zu sortierenden Werte $a_1\ a_2 \ldots a_n$ in einem festen Intervall [UNT, OB]: $UNT \le a_i \le OB$.
Der Einfachheit halber nehmen wir an, die a_i seien natürliche Zahlen und es seien UNT = 0 und OB = m-1.

Bucketsort: Verteile ("streue") die n Elemente A(1..n) auf m nacheinander angeordnete Fächer bucket(0..m-1) ("Eimer" genannt, daher "bucketsort"), hole sie anschließend in der Reihenfolge der Fächer wieder heraus und lege sie im Feld A ab.

Aufwand: O(n+m) sowohl für die Zeit als auch für den Platz.

Programm zum Sortieren (fügen Sie die Listenbearbeitung selbst hinzu):


```
A: array (1..n) of integer;
bucket: array (0..m-1) of "liste von integer"; ...
begin
for j in 0..m-1 loop bucket(j) := "leer"; end loop;
for i in 1..n loop                                  -- streuen
    "hänge A(i) an bucket(A(i)) an";
end loop;
i := 0;
for j in 0..m-1 loop                                -- sammeln
    while "bucket(j) nicht leer" loop
        i := i+1; A(i) := "erstes Element von bucket(j)";
        "Entferne aus bucket(j) das erste Element"; end loop;
end loop;
end;
```