6.1 Sortieren durch Aussuchen/Auswählen

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

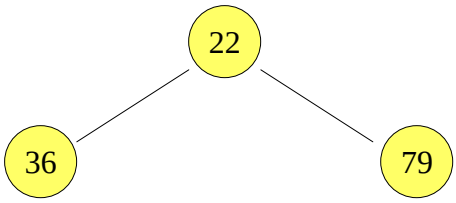
Mit dem Minimum sortieren:

```
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;
    A(pos) := A(i); A(i) := min;    -- nun steht das kleinste Element an Position i
end loop;
```

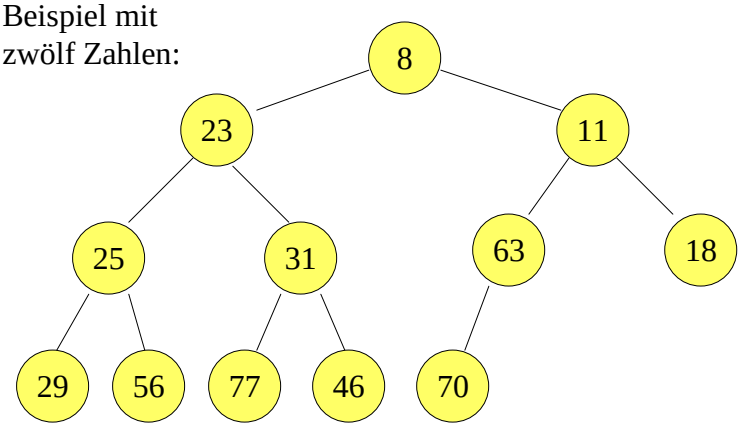
Zahl der Vergleiche stets $1/2 \cdot n \cdot (n-1)$ Schritte: $\Theta(n^2)$
Platzaufwand 4 Speicherplätze: $O(1)$

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

Ja, als binärer 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 absteigend 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 $A(i) \leq A(2i)$ und $A(i) \leq A(2i+1)$. Folgen oder Felder mit dieser Eigenschaft nennen wir "Heap" (meist übersetzt mit "Haufen"; sie haben nichts mit der Halde aus Kapitel 2 zu tun, die englisch ebenfalls heap heißt).

Definition:

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

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

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:

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

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

Heapsort:

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

1. Wandle dieses Feld in einen absteigenden Heap um:
   $A(1) \geq A(2) \geq A(3) \geq ... \geq A(n)$.
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.

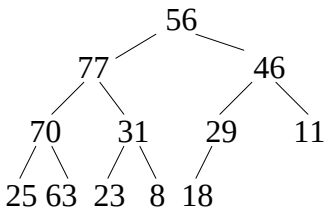
Im Folgenden beschreiben wir das Umwandeln in einen Heap (1.) und die Wiederherstellung der Heap-Eigenschaft (2.).

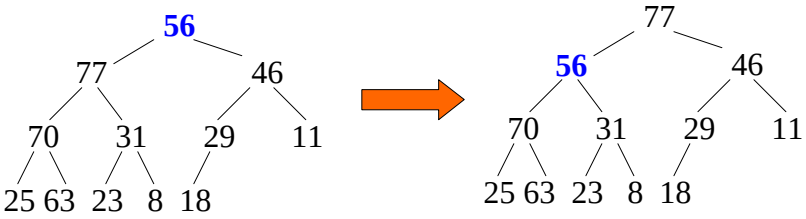
Die zentrale Prozedur ist die Herstellung der Heap-Eigenschaft in dem Teil des Feldes A, das mit dem Index links beginnt und mit dem Index rechts endet, unter der Annahme, dass höchstens beim Index links die Heap-Eigenschaft verletzt ist.

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".

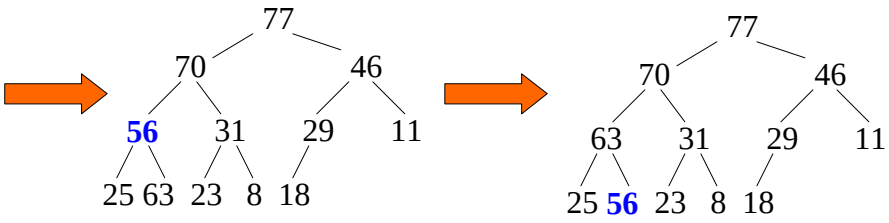
Betrachte ein Beispiel:

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.



```
procedure sink (links, rechts: 1..n) is          -- A und n sind global
i, j: natural;  weiter: Boolean;  v: <Elementtyp>;
begin  v := A(links); i := links; j := i+i;
    while  j <= rechts  loop
        if  j = rechts  then
            if A(j) > v then A(i):=A(j); A(j) := v;
            else A(i) := v; end if;  i := rechts;     -- i:=rechts  führt zum Abbruch
        elsif A(j) < A(j+1) then
                if v < A(j+1) then  A(i) := A(j+1); i := j+1;
                else A(i) := v; i := rechts; end if;
        else    if v < A(j) then A(i) := A(j); i := j;
                else A(i) := v; i := rechts; end if;
        end if;
        j := i+i;
    end loop;
end sink;
```

Durchläuft man die Schleife einmal, werden fast immer 2 Vergleiche zwischen Elementen durchgeführt.


```
procedure heapsort is                            -- A und n sind global
  procedure sink ... begin .... end sink;        -- siehe oben
  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)
$2 \cdot h/2 + 4 \cdot h/4 + 6 \cdot h/8 + 8 \cdot h/16 + ... + 2 \cdot \log(n) \cdot 1$
$= 2h \cdot (2 - 2 \cdot (\log(n)+1)/n) = 2 \cdot n - 2 \cdot \log(n) - 2 \leq 2 \cdot n$ 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 2.4. 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:

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

(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.

Diese Zahl wird aber auch im best case ungefähr erreicht, da Heapsort keine Vorsortierungen oder günstige Konstellationen ausnutzt und das Absinken meist (fast) bis zum Blatt geschieht. Dies deckt sich auch mit Experimenten. Somit erhalten wir den

Satz:

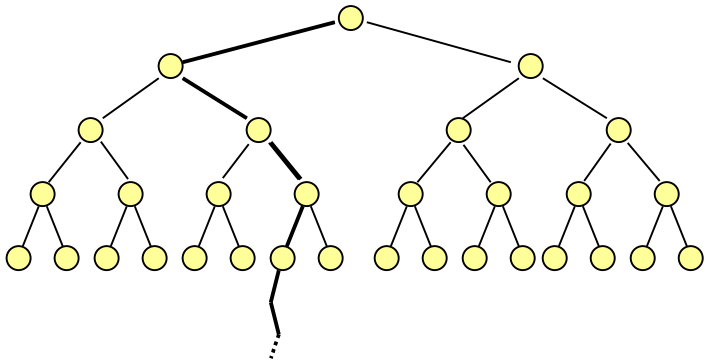
Dieses normale Heapsort (aus dem Jahre 1962) benötigt im schlimmsten Fall höchstens $2 \cdot n \cdot \log(n)$ Vergleiche.
Auch im besten Fall braucht man nicht wesentlich weniger Vergleiche.

Geht es nicht doch noch besser?

Ja, man kann den Faktor 2 noch verkleinern (allerdings kann er nicht kleiner als "1" werden, wie in Abschnitt 6.0 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".

Einsinkpfad: Dies ist der Weg, den ein Element, das in die Wurzel gesetzt wurde, nehmen muss, damit die Heap-Eigenschaft wiederhergestellt wird (vgl. auch Folie 8).
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.

```
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;
```


Bottom-up-Heapsort: (I. 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 < rechts  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 = rechts  then  A(j) := A(rechts); j := rechts;  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.
Hier gibt es keine Abschätzung im Mittel; Experimente bestätigen, dass der Faktor "2" vom normalen Heapsort auf "1" sinkt.

Insgesamt kann man 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 in der Praxis von $n \cdot \log(n) + O(n)$ Vergleichen ausgehen kann.

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

Beachte: Heapsort und seine Varianten sind garantierte n 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 (siehe Abschnitt 6.0) zu erreichen.