Lösungen zu „Fundamente der Informatik – Ablaufmodellierung, Algorithmen und Datenstrukturen“

Im Lehrbuch „Fundamente der Informatik – Ablaufmodellierung, Algorithmen und Datenstrukturen“ von Peter Hubwieser und Gerd Aiglstorfer (erschienen im Oldenbourg Verlag München, 2004, ISBN 3-486-27572-0) schließt jedes Kapitel mit einigen Aufgaben.

Die folgenden Lösungsvorschläge wurden von Gerd Aiglstorfer erstellt. Weitere Informationen und Hilfestellungen zum Lehrbuch finden Sie unter http://www.aigl.de.

Inhalt

1 Lösungsvorschläge zu „Sortieren und Suchen“

1.1 Lösungsvorschlag Aufgabe 11.1

Die Sortierung mittels „Sortieren durch Einfügen“

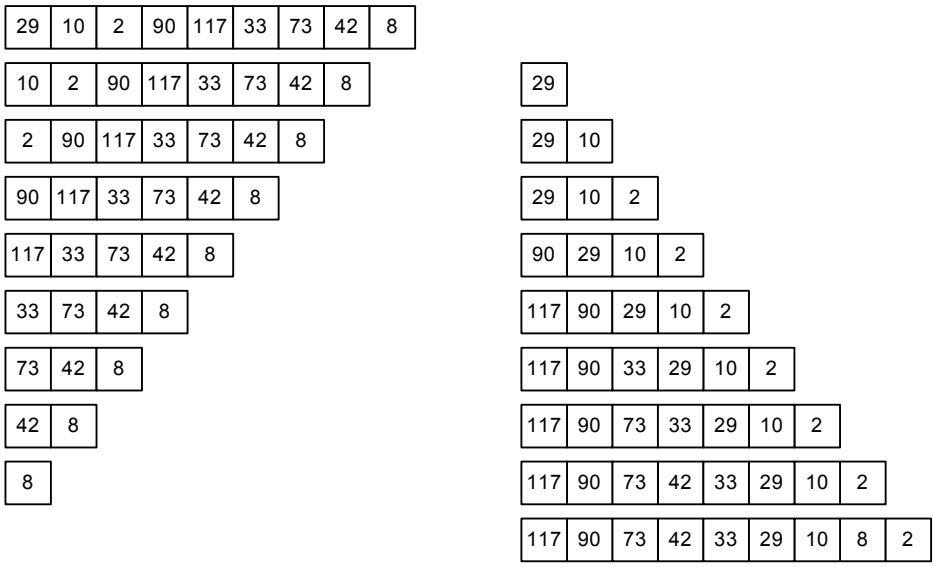

Abb. 1.1 *Sortieren durch Einfügen*

Die Sortierung mittels „Sortieren durch Auswählen“

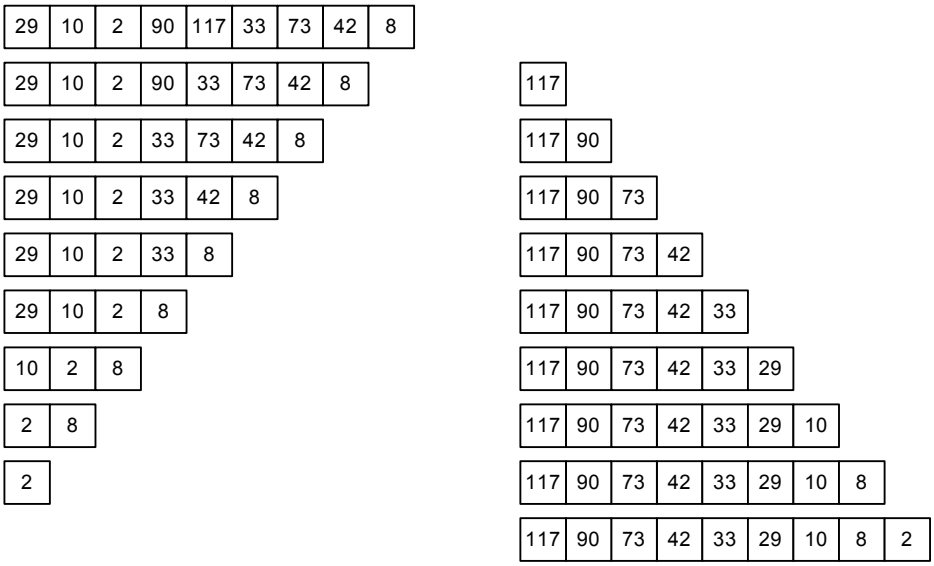

Abb. 1.2 *Sortieren durch Auswählen*

Die Sortierung mittels „Bubblesort“
Die folgenden Abbildungen zeigen jeweils einen Durchlauf der äußeren **while**-Schleife:

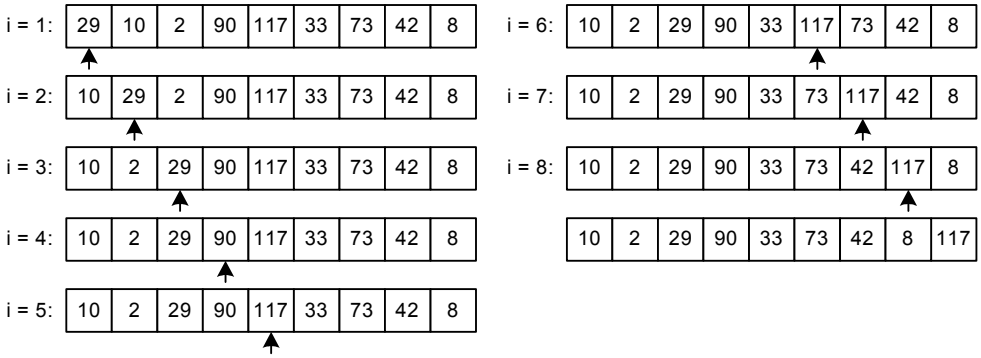

Abb. 1.3 *Sortieren mit bubblesort (1. Schleifendurchlauf)*

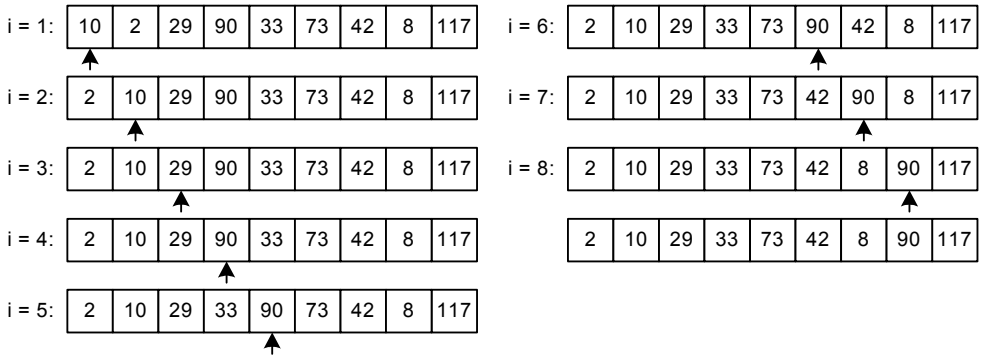

Abb. 1.4 *Sortieren mit bubblesort (2. Schleifendurchlauf)*

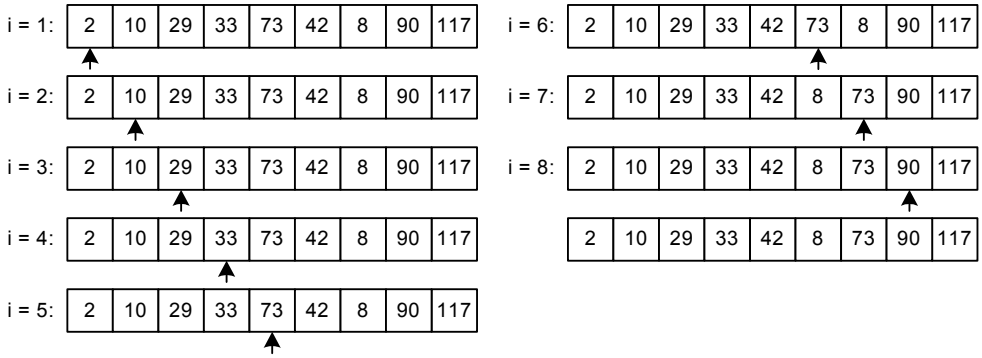

Abb. 1.5 *Sortieren mit bubblesort (3. Schleifendurchlauf)*

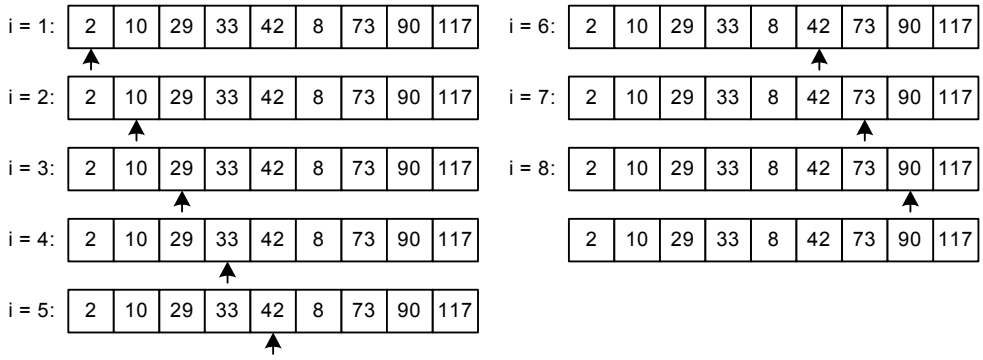

Abb. 1.6 *Sortieren mit bubblesort (4. Schleifendurchlauf)*

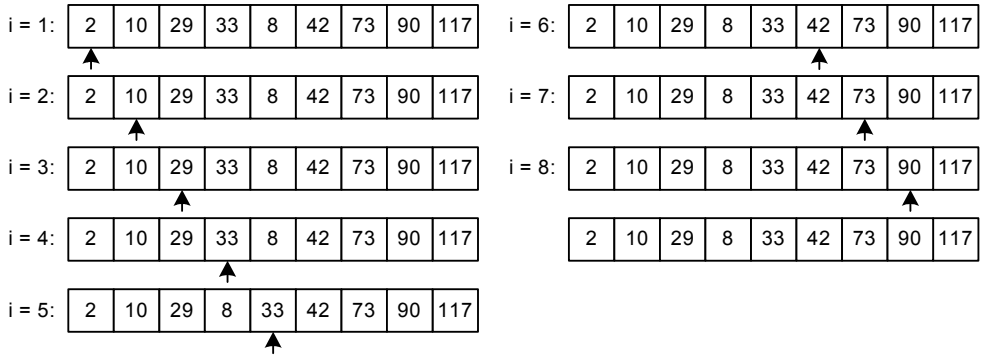

Abb. 1.7 *Sortieren mit bubblesort (5. Schleifendurchlauf)*

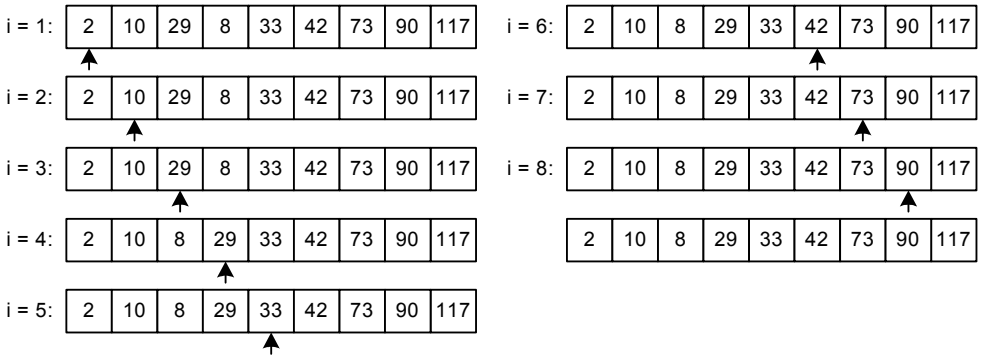

Abb. 1.8 *Sortieren mit bubblesort (6. Schleifendurchlauf)*

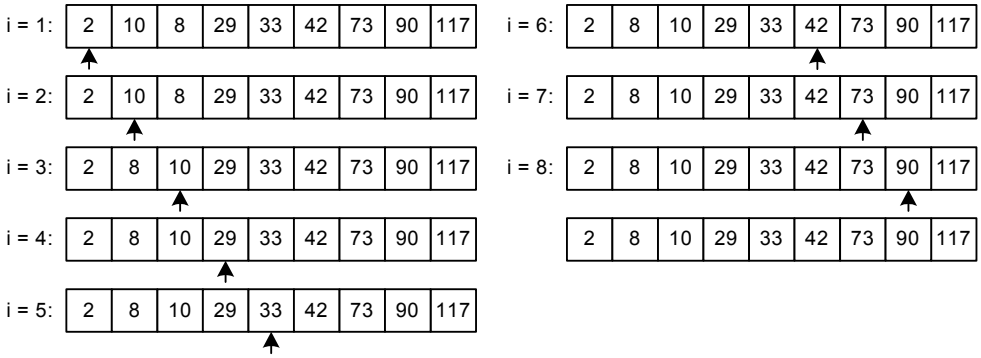

Abb. 1.9 *Sortieren mit bubblesort (7. Schleifendurchlauf)*

Die Sequenz ist zwar jetzt schon sortiert, jedoch wurde bei diesem Durchlauf noch eine Vertauschung vorgenommen. Daher wird ein weiterer Durchlauf der Schleife angestoßen.

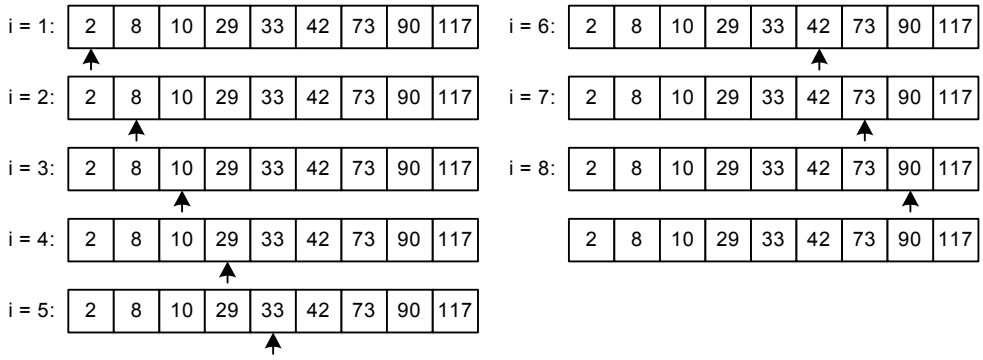

Abb. 1.10 *Sortieren mit bubblesort (8. und letzter Schleifendurchlauf)*

Die Sortierung mittels „Quicksort“

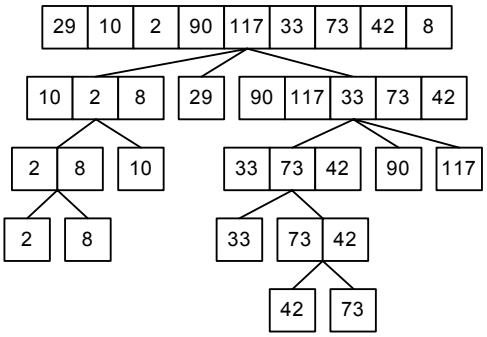

Abb. 1.11 *Sortieren mit quicksort*

Die Sortierung mittels „Heapsort“

Zuerst wird die Sequenz in einem vollständigen Binärbaum abgelegt, um danach mit Hilfe von *siftdown* beginnend mit dem Index *9/2 = 4* einen Heap zu erzeugen. An Position 4 muss nicht versickert werden (die Heap-Bedingung ist lokal bereits erfüllt), dafür aber an Position 3.

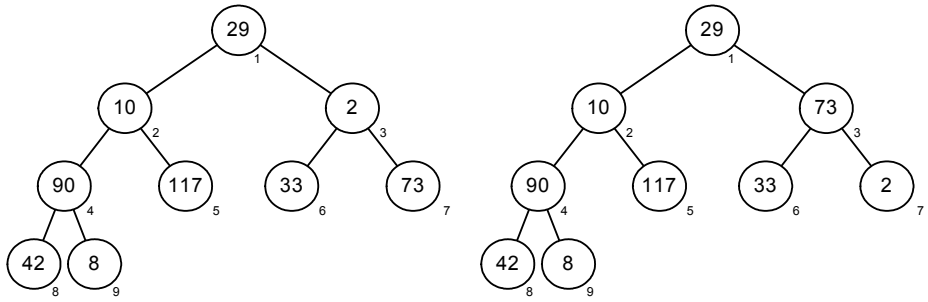

Abb. 1.12 *Sortieren mit heapsort (Erzeugen des Heaps: 1. Schritt)*

An Position 2 wird anschließend 10 und 117 getauscht (2. Schritt). Für den 3. und letzten Schritt zur Erzeugung des Heaps muss noch der Wert 29 an Index 1 versickert werden.

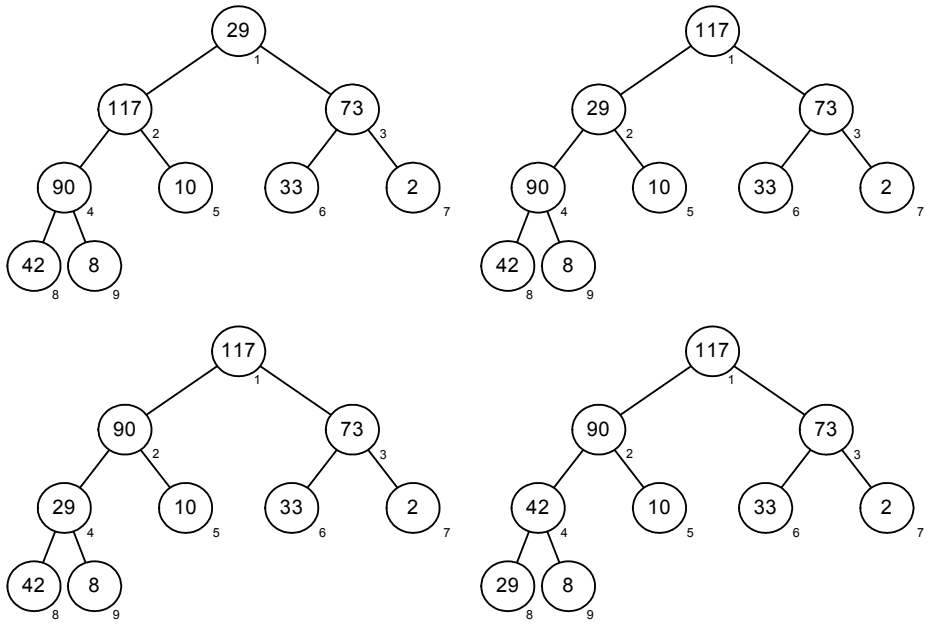

Abb. 1.13 *Sortieren mit heapsort (Erzeugen des Heaps: 2. und 3. Schritt)*

Für den Binärbaum rechts unten gilt nun die Heap-Bedingung und das Sortieren kann beginnen (2. Teil des *heapsort*-Algorithmus). Die folgenden Abbildungen zeigen den unsortierten Bereich der Sequenz als Heap in der Darstellung als Binärbaum (Baum links: Tausch von erstem und letztem noch nicht sortiertem Element, Bild rechts: Wiederherstellung der Heap-

Bedingung mit Hilfe von *siftdown*), wobei die Indizes der Bäume und Sequenzen jeweils übereinstimmen.

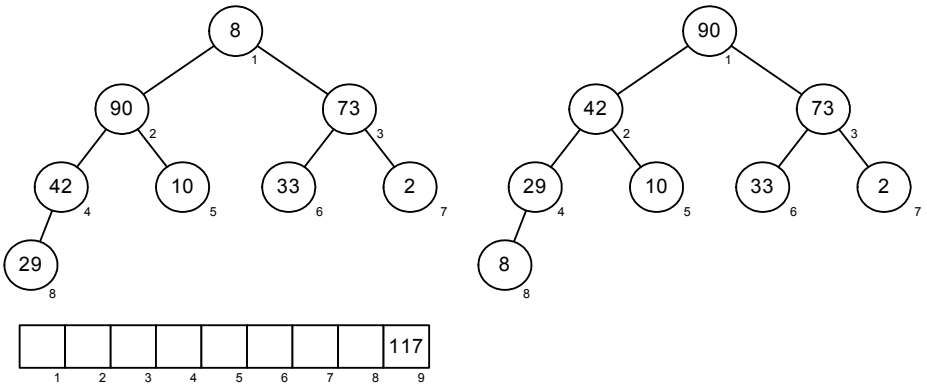

Abb. 1.14 *Sortieren mit heapsort (i := 9)*

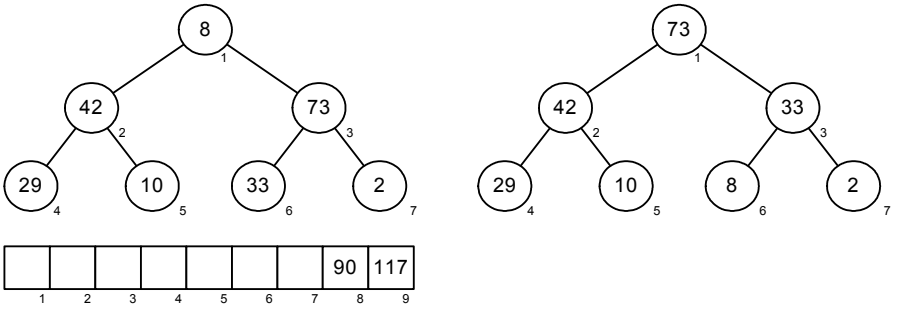

Abb. 1.15 *Sortieren mit heapsort (i := 8)*

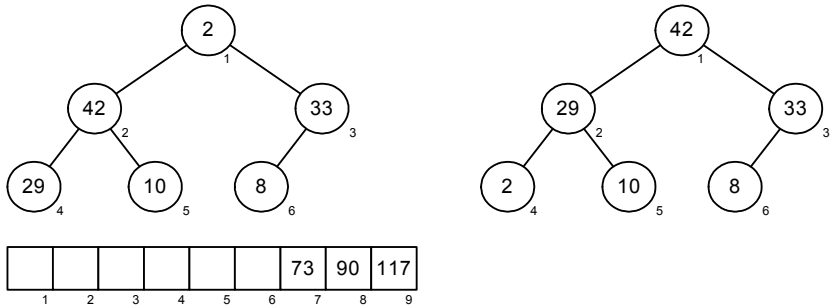

Abb. 1.16 *Sortieren mit heapsort (i := 7)*

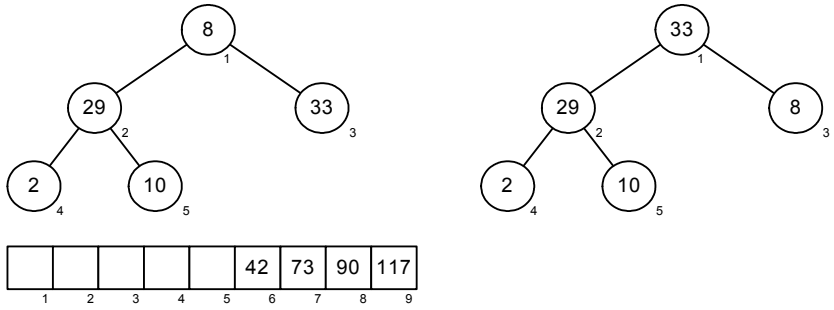

Abb. 1.17 *Sortieren mit heapsort (i := 6)*

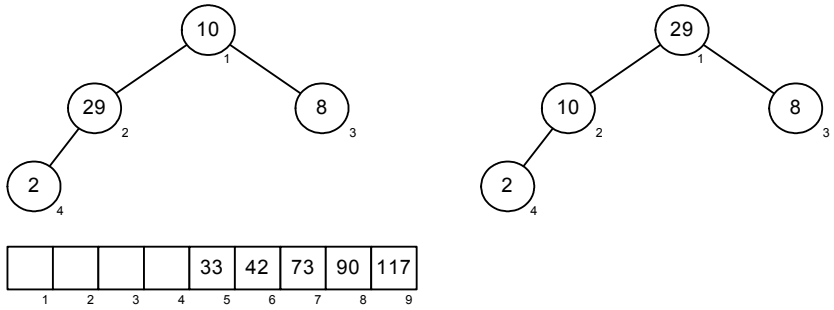

Abb. 1.18 *Sortieren mit heapsort (i := 5)*

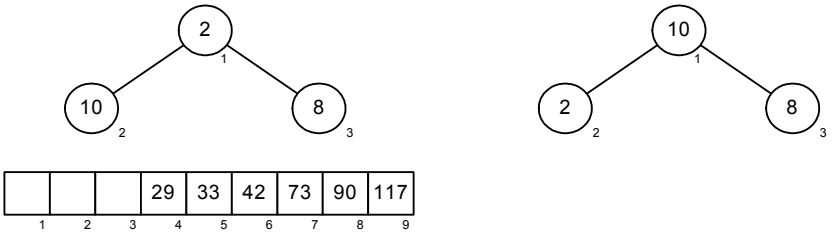

Abb. 1.19 *Sortieren mit heapsort (i := 4)*

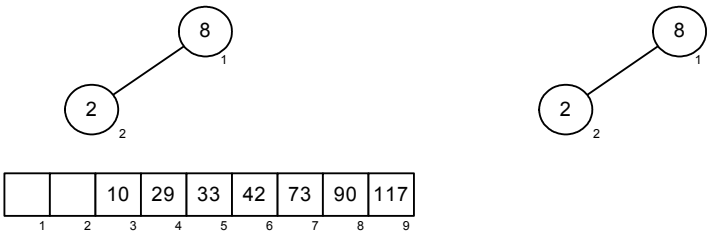

Abb. 1.20 *Sortieren mit heapsort (i := 3)*

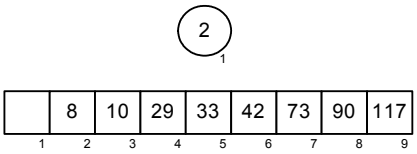

Abb. 1.21 *Sortieren mit heapsort (i := 2)*

1.2 Lösungsvorschlag Aufgabe 11.2

Zum aufsteigenden Sortieren durch Einfügen muss lediglich die Funktion *insert* aus Kapitel 11.1 angepasst werden:

```
function insert (seq nat s, nat a): seq nat
return if isempty(s) then ⟨a⟩
       elseif a ≤ first(s) then ⟨a⟩ ○ s
       else ⟨first(s)⟩ ○ insert(rest(s), a)
```

Zum Sortieren durch Auswählen wird die Funktion *selectmax* durch *selectmin* ersetzt und *appendseq* entsprechend angepasst:

```
function appendseq (seq nat s, r): seq nat
return if isempty(r) then s
       else
            let nat min = selectmin(rest(r), first(r)) in
            appendseq(s ∘ ⟨min⟩, delete(r, min))

function selectmin (seq nat s, nat a): nat
return if isempty(s) then a
       elseif a ≤ first(s) then selectmin(rest(s), a)
       else selectmin(rest(s), first(s))
```

Eine weitere Möglichkeit wäre die folgende (nur *appendseq* wird verändert):

```
function appendseq (seq nat s, r): seq nat
return if isempty(r) then s
       else
            let nat max = selectmax(rest(r), first(r)) in
            appendseq(⟨max⟩ ∘ s, delete(r, max))
```

1.3 Lösungsvorschlag Aufgabe 11.3

Im Falle von *bubblesort* ist nur der Vergleich der benachbarten Elemente der Sequenz *s* zu verändern (beim ersten Durchlauf der inneren **while**-Schleife wird das kleinste Element an das Ende der Sequenz geschoben):

```
procedure bubblesort (var seq nat s):
var nat i;
var bool nichtvertauscht := false;
begin
while not nichtvertauscht do
      i := 1;
      nichtvertauscht := true;
      while i ≤ n-1 do
            if s[i] < s[i+1] then
                  s[i], s[i+1] := s[i+1], s[i];
                  nichtvertauscht := false
            endif
            i := i + 1
      endwhile
endwhile
endproc
```

1.4 Lösungsvorschlag Aufgabe 11.4

Teilaufgabe a)
Wir führen eine Variable *j* ein, damit mit jedem Durchlauf der äußeren **while**-Schleife die Grenze zwischen nicht sortiertem und sortiertem Bereich verschoben wird:

```
procedure bubblesort (var seq nat s):
var nat i, j := 1;
var bool nichtvertauscht := false;
begin
while not nichtvertauscht do
      i := 1;
      nichtvertauscht := true;
      while i ≤ n-j do
            if s[i] > s[i+1] then
                  s[i], s[i+1] := s[i+1], s[i];
                  nichtvertauscht := false
            endif
            i := i + 1
      endwhile
      j := j + 1
endwhile
endproc
```

Teilaufgabe b)
Im ersten Schleifendurchlauf benötigt *bubblesort n–1* Vergleiche und mit jedem weiteren wird es ein Vergleich weniger. Somit gilt:

$$T_{\text{bubblesort}} = (n-1)+(n-2)+(n-3)+\ldots+2+1 = \sum_{i=1}^{n-1} i = \frac{(n-1)\cdot(n-1+1)}{2} = O(n^2).$$

Die Optimierung hat also keine Verbesserung der worst case-Laufzeit zur Folge.

1.5 Lösungsvorschlag Aufgabe 11.5

Die vier Funktionen werden wie folgt angegeben:

```
function lowerpart (seq nat s, nat a): seq nat
return if isempty(s) then empty
       elseif a > first(s) then
            ⟨first((s)⟩ ∘ lowerpart(rest(s), a)
       else lowerpart(rest(s), a)
```

```
function equalpart (seq nat s, nat a): seq nat
return if isempty(s) then empty
       elseif a = first(s) then
            ⟨a⟩ ∘ equalpart(rest(s), a)
       else equalpart(rest(s), a)

function higherpart (seq nat s, nat a): seq nat
return if isempty(s) then empty
       elseif a < first(s) then
            ⟨first((s)⟩ ∘ higherpart(rest(s), a)
       else higherpart(rest(s), a)

function length (seq nat s): nat
return if isempty(s) then 0
       else 1 + length(rest(s))
```

1.6 Lösungsvorschlag Aufgabe 11.6

Um mit *quicksort* aufsteigend zu sortieren, wird der **else**-Zweig umgestellt:

```
function quicksort (seq nat s): seq nat
return if length(s) ≤ 1 then s
       else quicksort(lowerpart(s, first(s))) ∘
            equalpart(s, first(s)) ∘
            quicksort(higherpart(s, first(s)))
```

1.7 Lösungsvorschlag Aufgabe 11.7

Teilaufgabe a)
Eine Möglichkeit zur Berechnung des Pivot-Elements ist die Bildung des Mittelwerts über alle Elemente in der Sequenz. Also:

$$\lfloor (a_1 + a_2 + \ldots + a_n)/n \rfloor \text{ oder } \lceil (a_1 + a_2 + \ldots + a_n)/n \rceil.$$

Dieses Verfahren liefert zwar einen besseren Wert als die Wahl des ersten Elements der Sequenz, hat aber den Nachteil, dass „Ausreißer“ unter den Werten eine Berechnung des optimalen Pivot-Elements verhindern. Die Sequenz 15, 3, 2, 100, 8, 11 hat den Mittelwert:

$$\lfloor (15 + 3 + 2 + 100 + 8 + 11)/6 \rfloor = 23.$$

Das Pivot-Element 23 würde die Sequenz in zwei Teilsequenzen mit den Längen 5 und 1 teilen. Der optimale Wert liegt jedoch zwischen 8 und 11 (es würden zwei Teilsequenzen gleicher Länge entstehen). Diese Art der Bestimmung des Pivot-Elements liefert also nur gute Werte, wenn die zu sortierenden Werte im Wertebereich gleich verteilt sind.

Teilaufgabe b)
Anforderungen an ein optimal gewähltes Pivot-Element:

- Das Pivot-Element soll die Sequenz *s* in zwei gleich große Teilsequenzen teilen.
- Es soll einfach und effizient zu berechnen sein. Optimal wäre mit der Komplexität *O(1)*, also mit konstanter Laufzeit.

1.8 Lösungsvorschlag Aufgabe 11.8

Die Minimum-Heap-Bedingung definieren wir für vollständige Binärbäume wie folgt:

> Jeder Schlüssel eines Knotens ist kleiner oder gleich
> den Schlüsseln der vorhandenen Kinderknoten.

Für die Definition des Heaps als Folge $F = k_1, k_2, \ldots, k_n$ bedeutet dies: $k_i \le k_{2i}$ und $k_i \le k_{2i+1}$ für $2i$, $2i+1 \le n$ oder $k_i \ge k_{\lfloor i/2 \rfloor}$, sofern $2 \le i \le n$.

Die Prozedur *siftdown* wird damit für absteigende Sortierung angepasst. Beim Versickern ist das kleinere Kind für den Vergleich mit dem Vaterknoten zu wählen (erste **if**-Anweisung). Der Tausch erfolgt nur, wenn der Wert im Vaterknoten größer als der Wert des Kindknotens ist (zweite **if**-Anweisung: das kleinere Element wird in den Vaterknoten verschoben):

```
procedure siftdown (var seq nat s, var nat i, nat m):
var nat j;
begin
while 2i ≤ m do
      j := 2i;
      if j < m and s[j] > s[j+1] then j := j + 1 endif
      if s[i] > s[j] then
            s[i], s[j] := s[j], s[i];
            i := j
      else i := m
      endif
endwhile
endproc
```

heapsort muss nun für absteigendes Sortieren nicht mehr verändert werden.

1.9 Lösungsvorschlag Aufgabe 11.9

Alle behandelten Sortierverfahren sortieren Sequenzen mit identischen Zahlen. Begründung:

- Sortieren durch Einfügen: *insert* fügt identische Zahlen nebeneinander in die sortierte Sequenz ein.

- Sortieren durch Auswählen: *selectmax* selektiert gleiche Elemente nacheinander und *delete* löscht bei jedem Aufruf nur ein Vorkommen des Elements in der noch nicht sortierten Sequenz.
- Bubblesort: Befinden sich mehrere, gleiche Elemente in der Sequenz, so werden diese soweit in der Sortierreihenfolge nach rechts geschoben, bis sie nebeneinander stehen (die identischen Elemente werden nicht untereinander vertauscht).
- Quicksort: *equalpart* erzeugt eine Sequenz, welche mehr als ein Element aufnehmen kann.
- Heapsort: die Heap-Bedingung sorgt dafür, dass gleiche Elemente immer nebeneinander im vollständigen Binärbaum stehen und somit auch nacheinander zum Sortieren in die Wurzel gelangen.

1.10 Lösungsvorschlag Aufgabe 11.10

Die Suche mittels „sequentieller Suche“

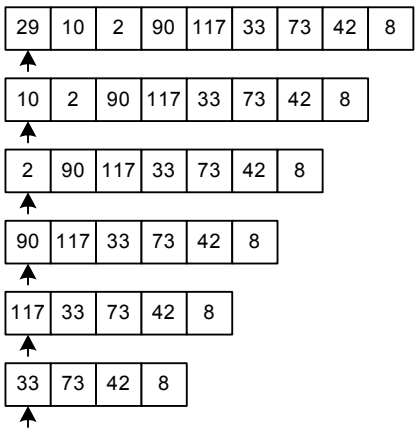

Abb. 1.22 *Sequentielle Suche nach k = 33*

Die Suche mittels „binärer Suche“

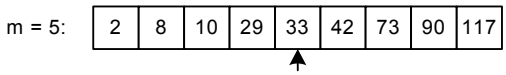

Abb. 1.23 *Binäre Suche nach k = 33*

Die Suche mittels „Interpolationssuche“
Berechnung der geschätzten Position von *k = 33* in der sortierten Sequenz 2, 8, 10, 29, 33, 42, 73, 90, 117:

$$m = 1 + \left\lfloor \frac{33 - 2}{117 - 2} \cdot (9 - 1) \right\rfloor = 3.$$

Die Schlüsselverteilung hat in diesem Fall zur Folge, dass Interpolationssuche nicht so gut arbeitet wie binäre Suche. Wir berechnen für *l = 4* und *r = 9* weiter:

$$m = 4 + \left\lfloor \frac{33 - 29}{117 - 29} \cdot (9 - 4) \right\rfloor = 4.$$

Und nun mit *l = 5* und *r = 9*:

$$m = 5 + \left\lfloor \frac{33 - 33}{117 - 33} \cdot (9 - 5) \right\rfloor = 5.$$

Mit der Verwendung von $\lceil . \rceil$ an der Stelle von $\lfloor . \rfloor$ wäre eine Berechnung von *m* weniger nötig gewesen.

Die Suche mittels „Binärbaumsuche“
Die zu durchsuchende Zahlenfolge wird im binären Suchbaum *b* abgelegt (die Verteilung ist beliebig, sie könnte auch anders sein):

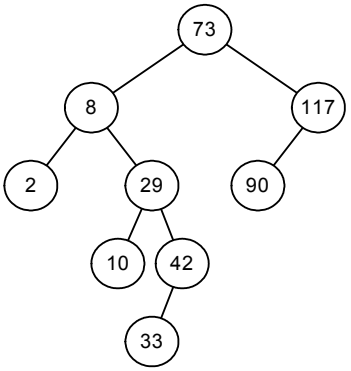

Abb. 1.24 *Binärer Suchbaum*

Der Baum hat die Sorte **bintree nat** und über die Definition des abstrakten Datentyps *Binarytree* die Form (es werden nur für die Suche entscheidenden Teilbäume dargestellt):

```
b = maketree(t1, 73, t2)
t1 = maketree(t3, 8, t4)
t2 = …
t3 = …
t4 = maketree(t5, 29, t6)
t5 = …
t6 = maketree(t7, 42, empty)
t7 = maketree(empty, 33, empty)
```

Der Aufruf lautet dann:

```
binarytree(b, 33)
```

Und es ergibt sich die Aufruffolge:

```
33 < 73 → binarytree(t1, 33)
33 > 8  → binarytree(t4, 33)
33 > 29 → binarytree(t6, 33)
33 < 42 → binarytree(t7, 33)
```

Der letzte Aufruf wird zu **true** ausgewertet.