Aufgabe 1

Wir geben zu jeder Teilaufgabe die Auflösung der Rekursionsgleichung, die Darstellung mit Hilfe der O-Notation und den Induktionsschritt des Beweises an; der Induktionsanfang ist jeweils offensichtlich.

(a)

$T(n) = 2\ (n\text{-}1)$
$T(n) = \mathrm{O}(n)$

$n \rightarrow 2n$: $T(2n) = T(n) + 2\ n = 2\ (n\text{-}1) + 2\ n = 2\ (2n - 1)$

(b)

$T(n) = 2^n\text{-}1$
$T(n) = \mathrm{O}(2^n)$

$n \rightarrow n+1$: $T(n+1) = 2\ T(n) + 1 = 2\ (2^n\text{-}1) + 1 = 2^{n+1}\ \text{-}1$

(c)

$$T(n) = \sum_{i=0}^{n} i = \frac{1}{2} \cdot n \cdot (n+1)$$

$n \rightarrow n+1$: $T(n+1) = 2\ T(n) - T(n\text{-}1) + 1 = n(n+1) - \frac{1}{2}(n\text{-}1)n + 1 = \frac{1}{2}(n^2 + 3n + 2) = \frac{1}{2}(n+1)(n+2)$

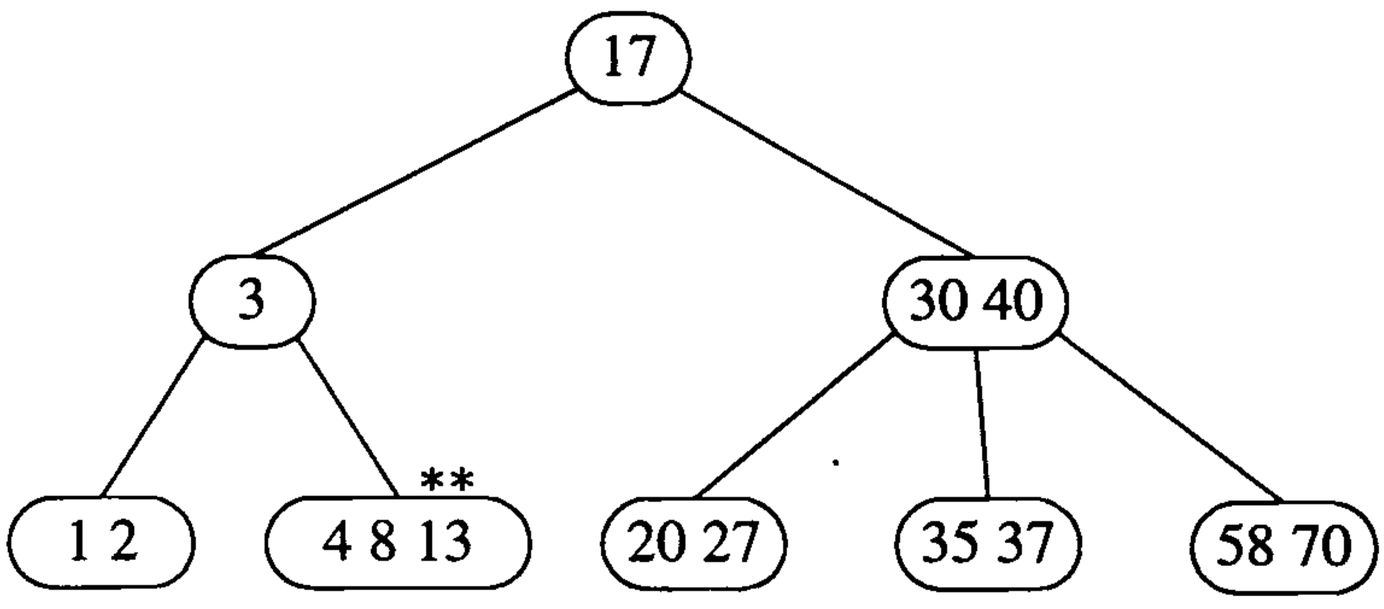
Aufgabe 2

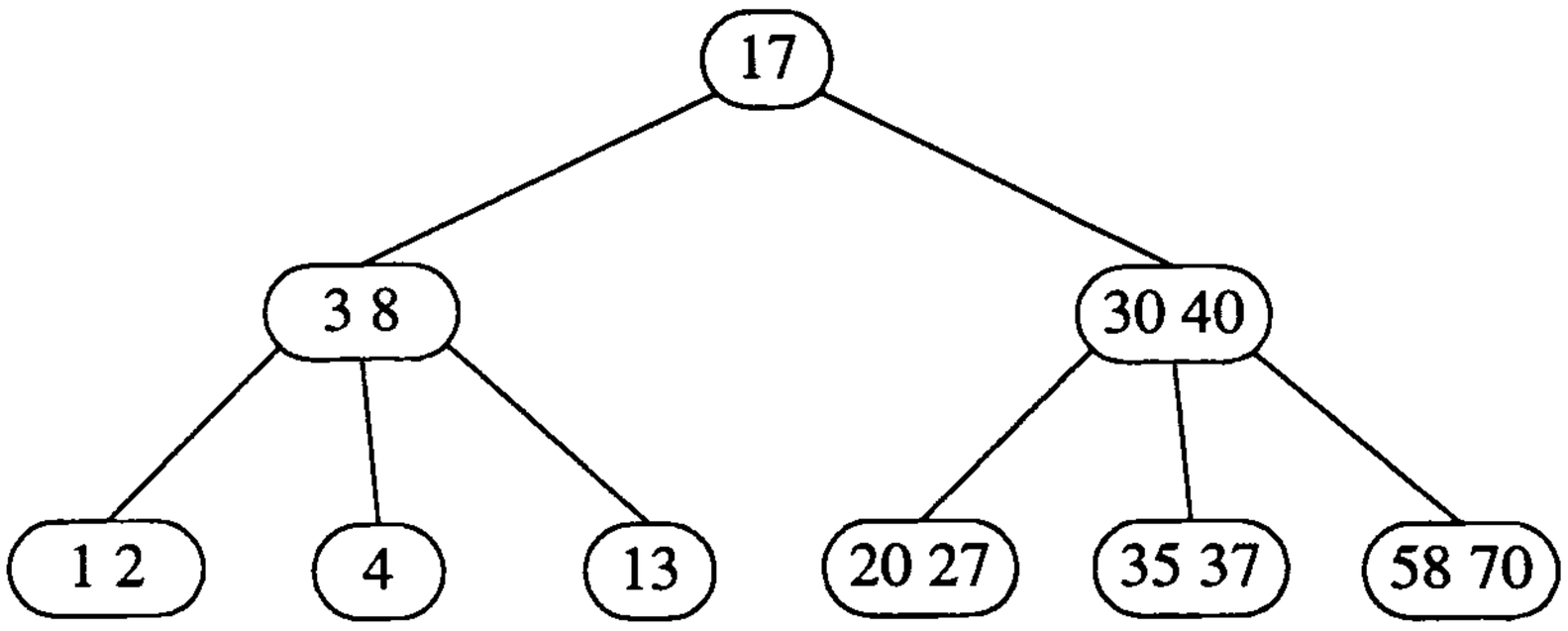
(a)

Knoten, bei denen eine Overflow-Operation durchgeführt wird, sind mit ** markiert.

Einfügen von 70: kein Overflow

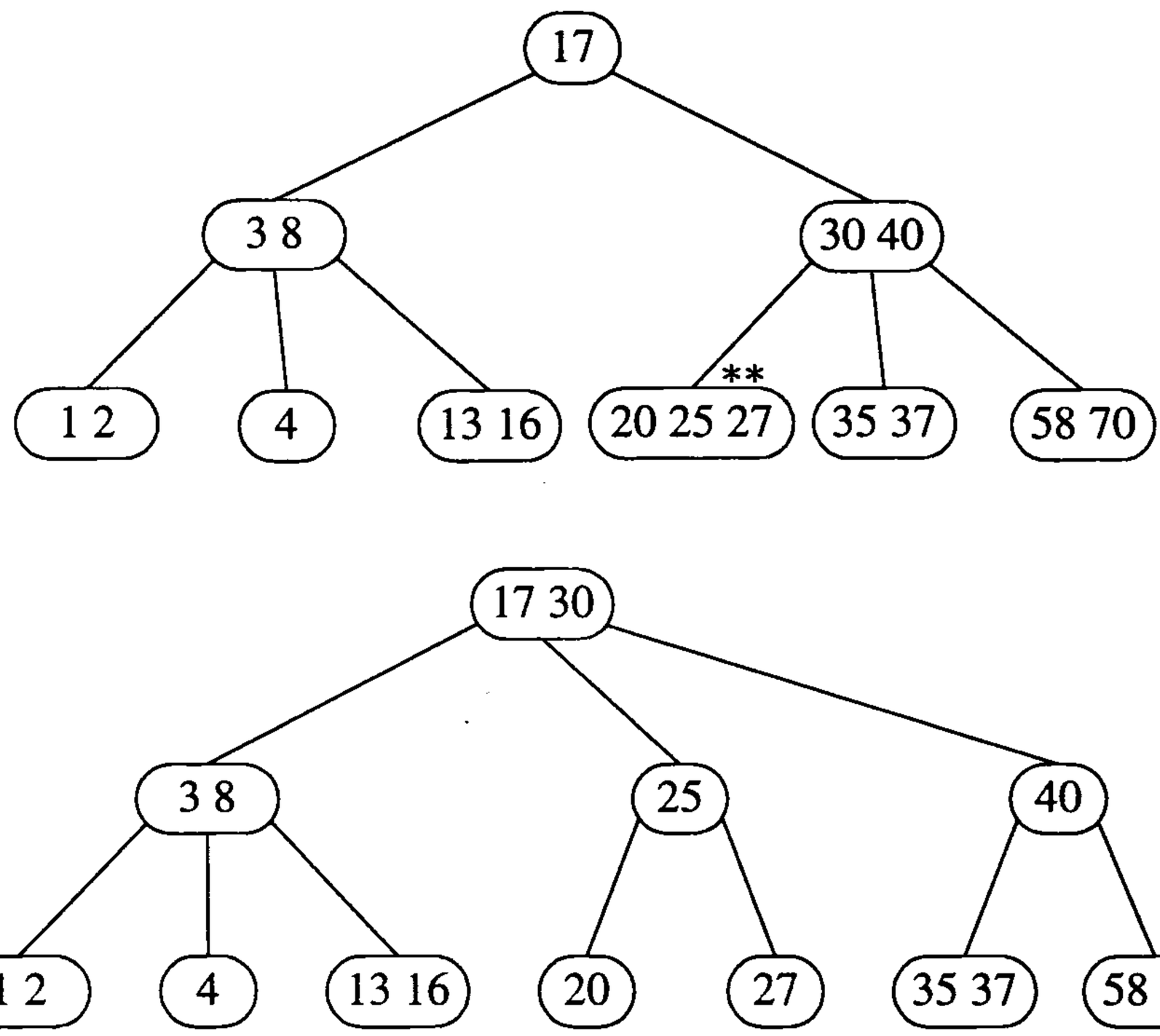
Einfügen von 8:





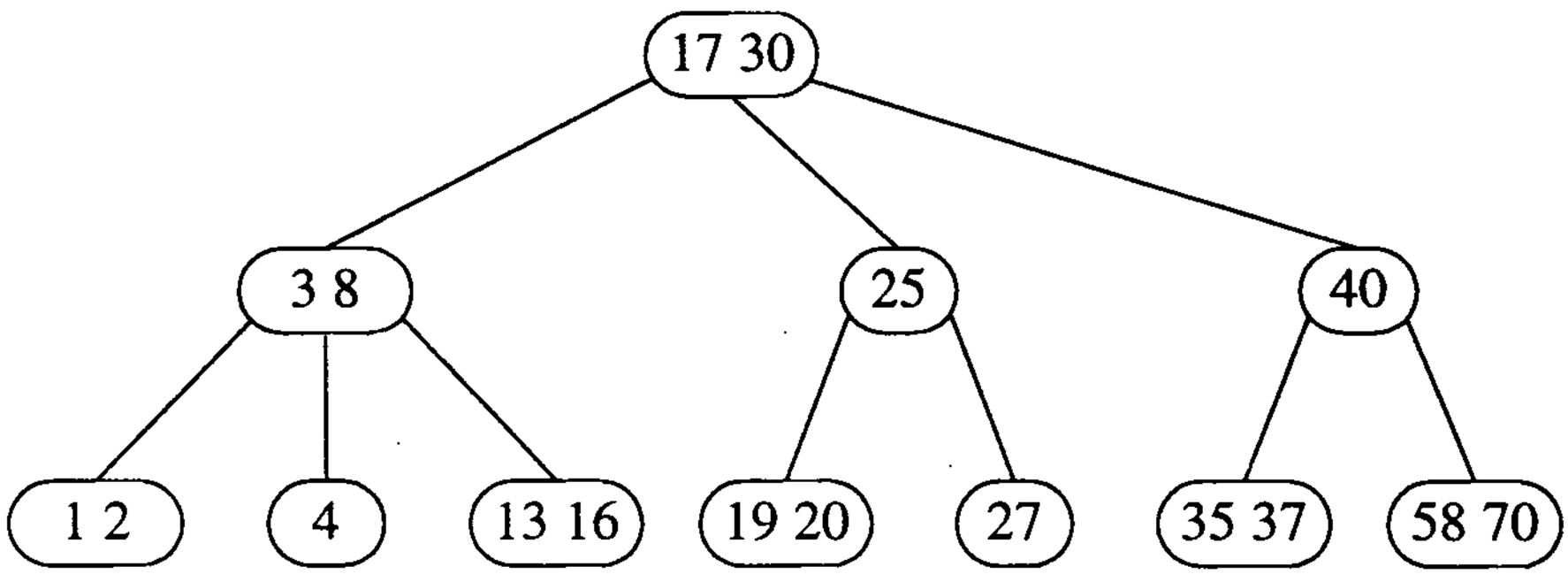
Einfügen von 16: kein Overflow

Einfügen von 25:



Einfügen von 19: kein Overflow

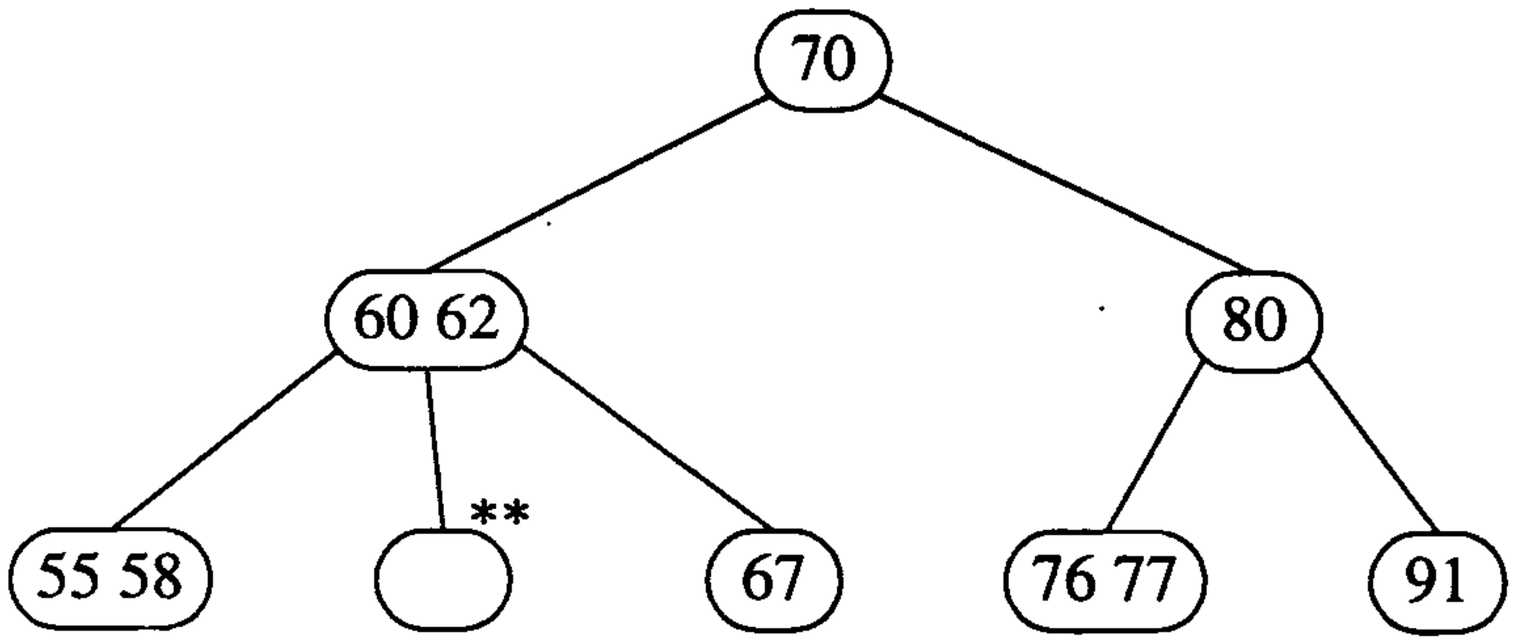
Ergebnisbaum nach dem Einfügen aller Schlüsselelemente:

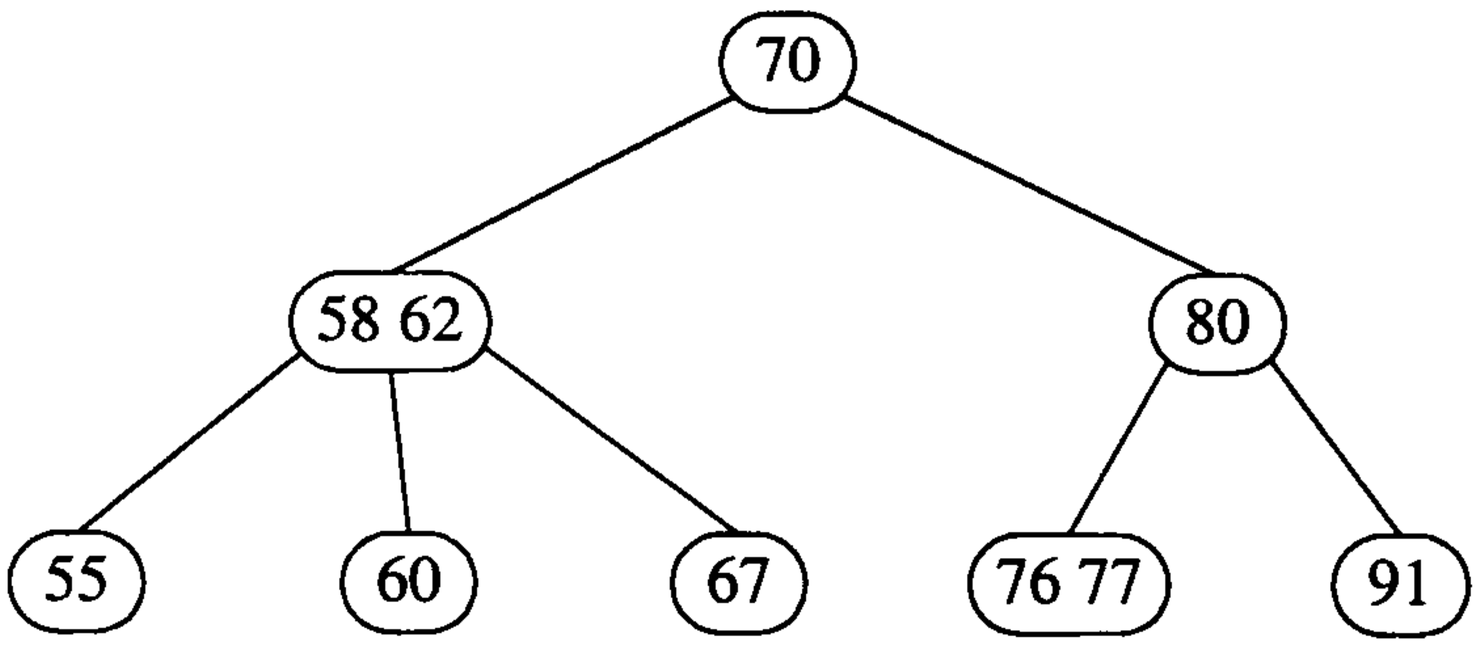


(b)

Knoten, bei denen eine Underflow-Operation durchgeführt wird, sind mit ** markiert.

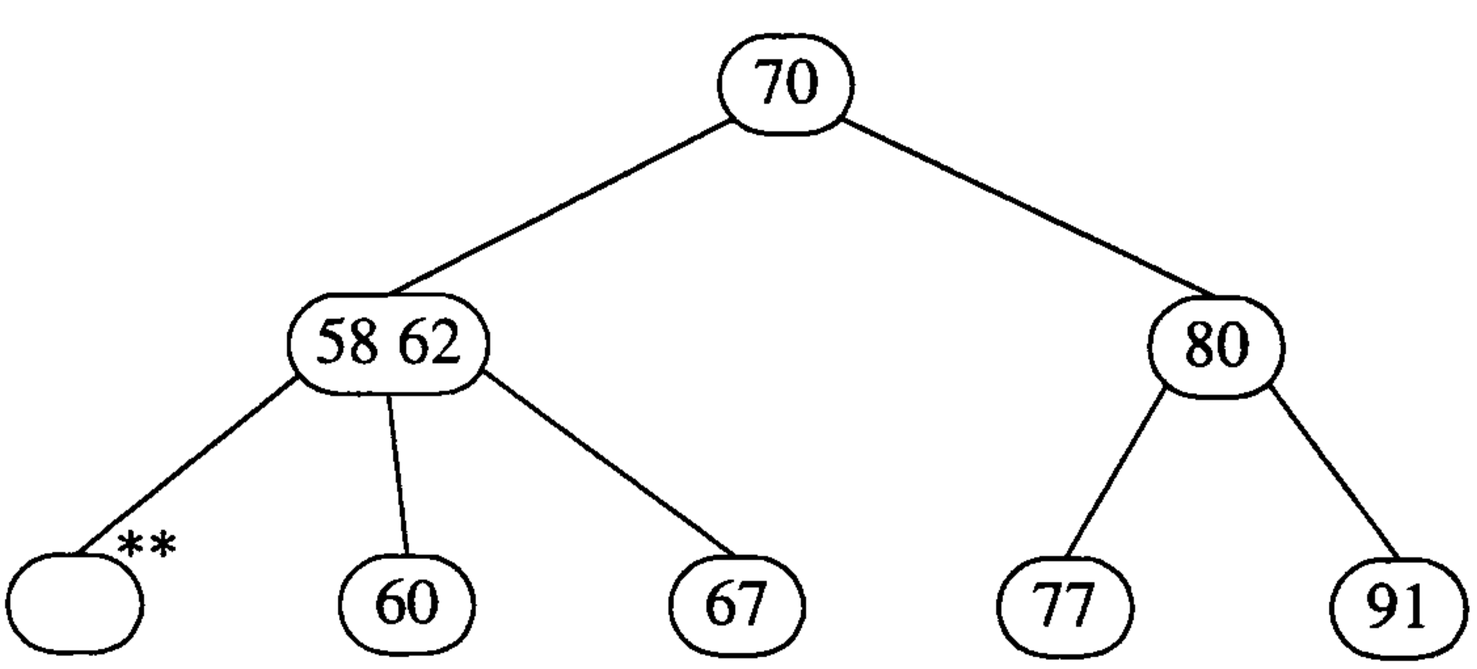
Löschen von 61 (Balance):





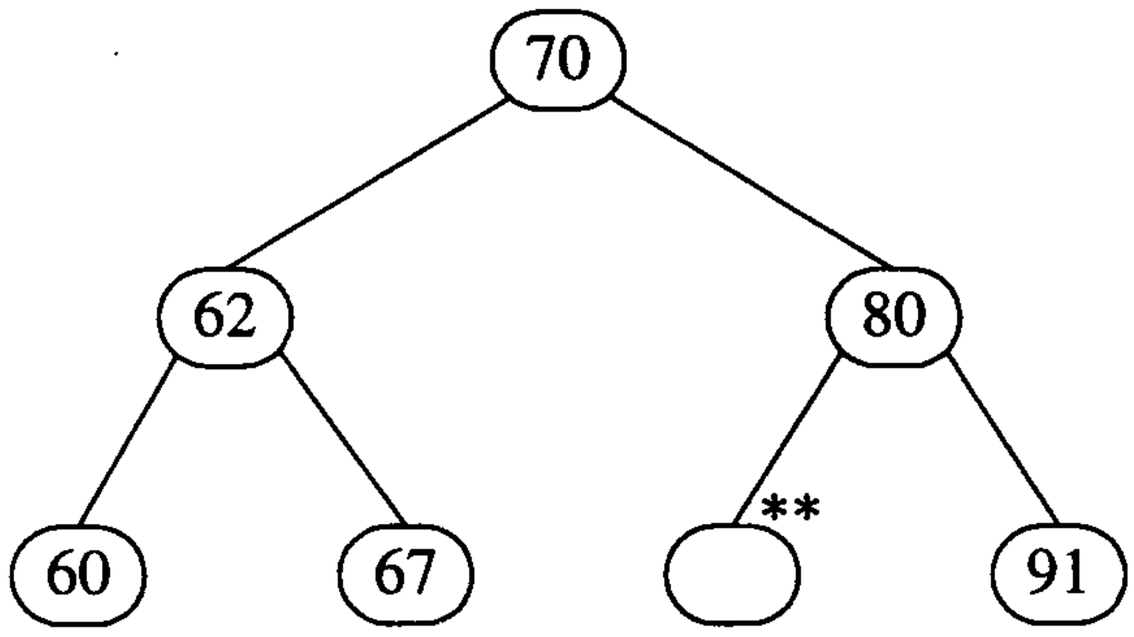
Löschen von 76: keine Underflow-Operation

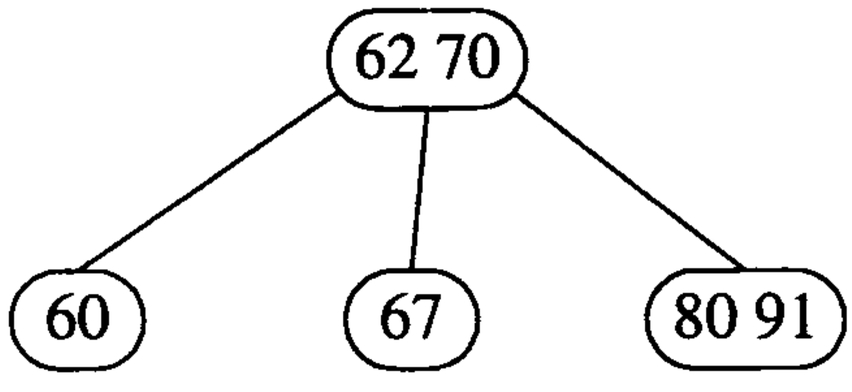
Löschen von 55 (merge):



Löschen von 58: keine Underflow-Operation

Löschen von 77 (merge):





Aufgabe 3

Ausgangsfolge: 35 62 28 50 11 45

nach $i = 2$: <u>35 62</u> 28 50 11 45

nach $i = 3$: <u>28 35 62</u> 50 11 45

nach $i = 4$: <u>28 35 50 62</u> 11 45

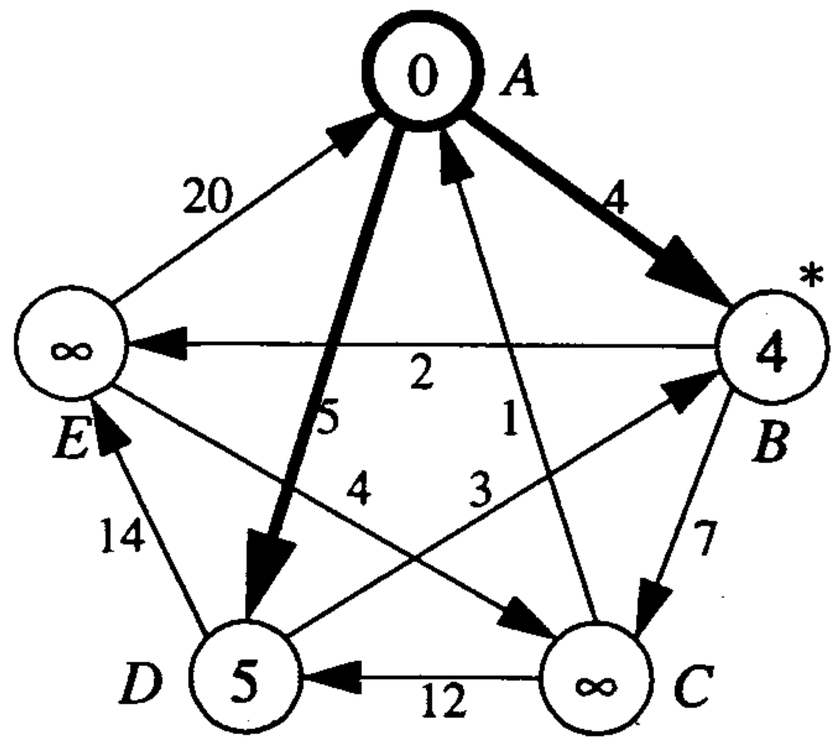
nach $i = 5$: <u>11 28 35 50 62</u> 45

nach $i = 6$: <u>11 28 35 45 50 62</u>

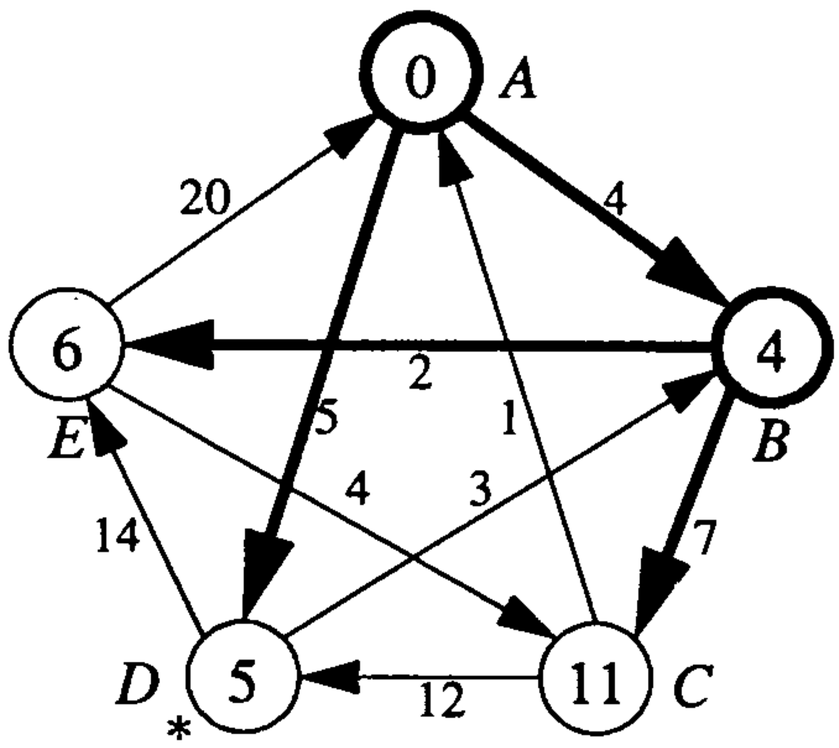
Aufgabe 4

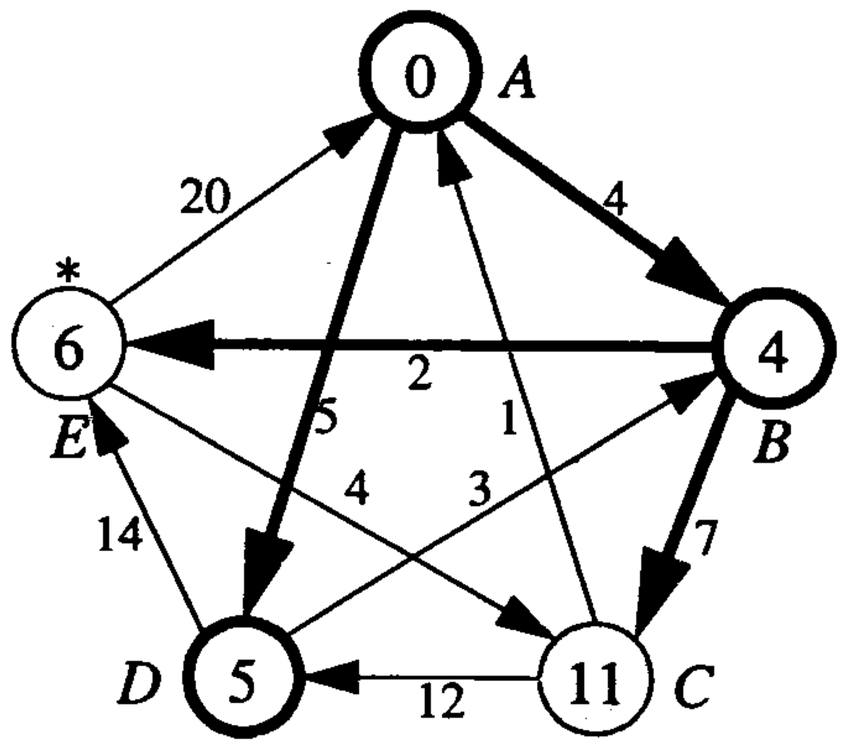
(a)

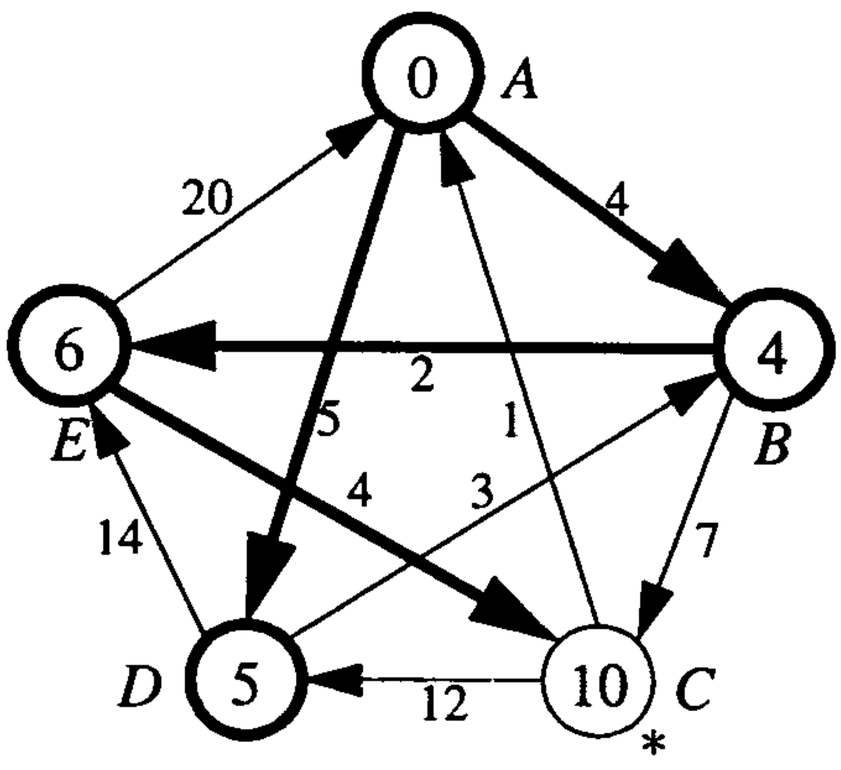
Der Knoten *A* wird zunächst mit 0 markiert. Alle anderen Knoten erhalten ∞. *A* wird als erstes fett gezeichnet (grün gefärbt):

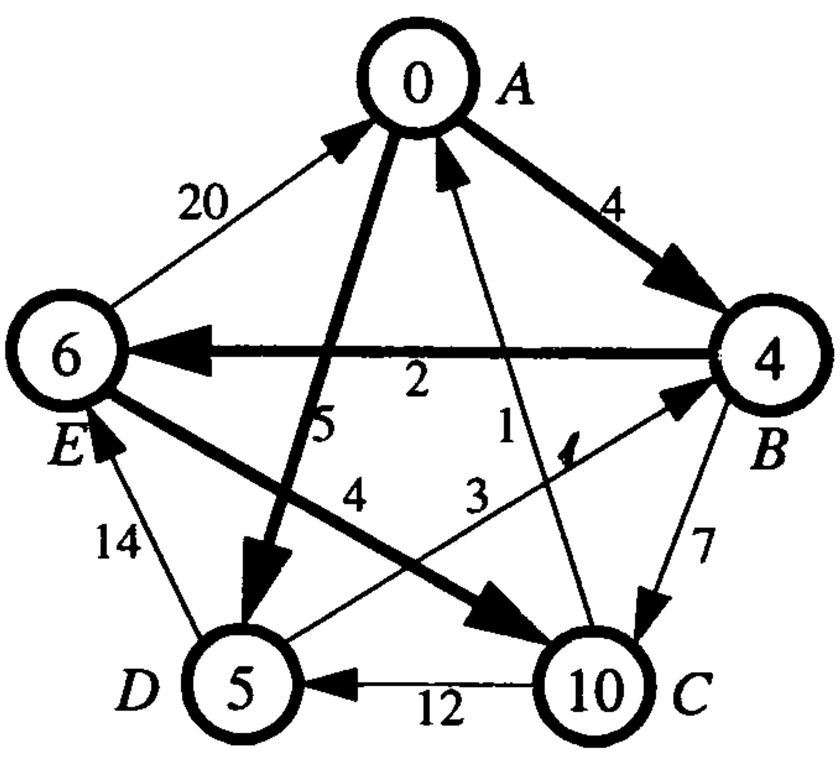


Der Reihe nach werden nun die Knoten B, D, E und C bearbeitet:









(b)

Wenn Dijkstras Algorithmus mit Adjazenzlisten und als Heap dargestellter Priority Queue implementiert wird, ist der Gesamtaufwand $O(e \log n)$, wobei e die Anzahl der Kanten des bearbeiteten Graphen ist. Der Platzbedarf ist $O(n + e)$.

Aufgabe 5

(a)

Es sei eine Menge S horizontaler und vertikaler Segmente gegeben. Die horizontalen Segmente sind durch ihre linken und rechten Endpunkte dargestellt. Der rekursive Algorithmus wird durch *linsect* aufgerufen. Die drei Schritte des Algorithmus sehen dann wie folgt aus:

Divide:

Wähle eine x-Koordinate x_m, die S in zwei etwa gleich große Teilmengen S_1 und S_2 zerlegt.

Conquer:

linsect (S_1); *linsect* (S_2); { *linsect* wird für jede der Teilmengen aufgerufen. Ein Aufruf von *linsect* hat nach der Beendigung alle Schnitte zwischen den in S_i repräsentierten Segmenten ausgegeben. Dabei ist ein horizontales Segment in S_i repräsentiert, wenn mindestens einer der Endpunkte in S_i enthalten ist.}

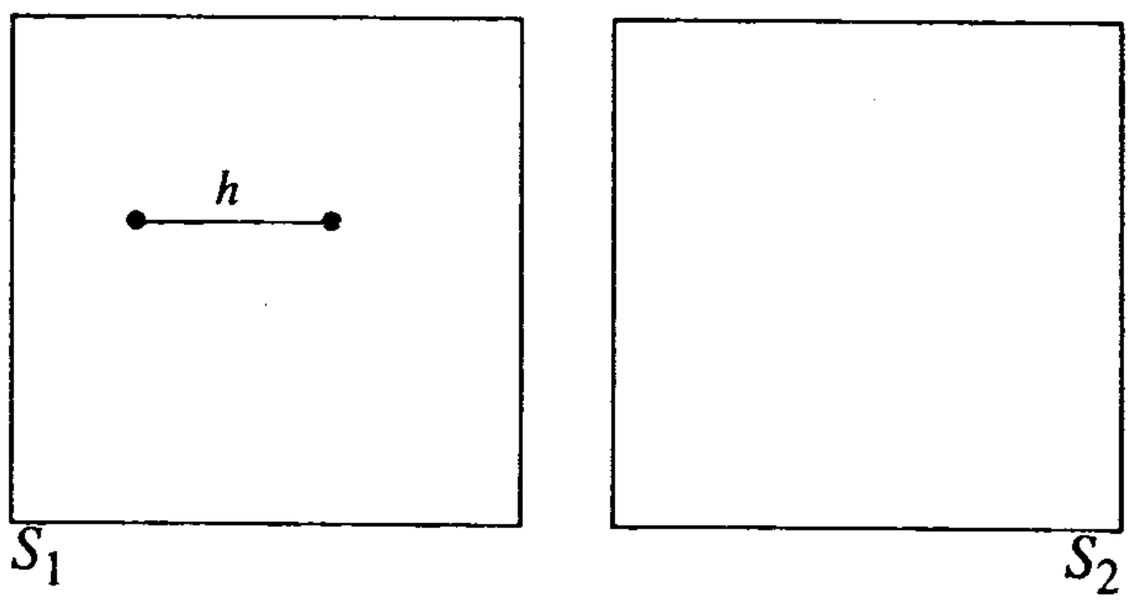
Merge:

Berechne die Schnitte zwischen horizontalen Segmenten in S_1 und vertikalen Segmenten in S_2 und umgekehrt.

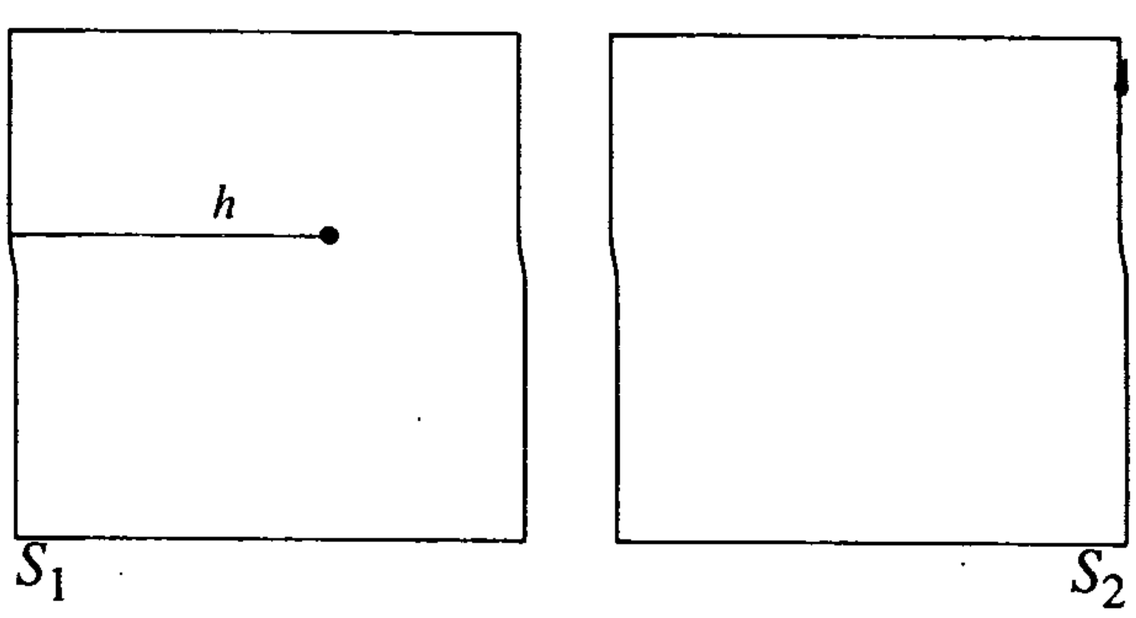
(b)

Im Folgenden werden die unterschiedlichen Fälle dargestellt. Wir betrachten ein einzelnes horizontales Segment h, das in S_1 repräsentiert ist.

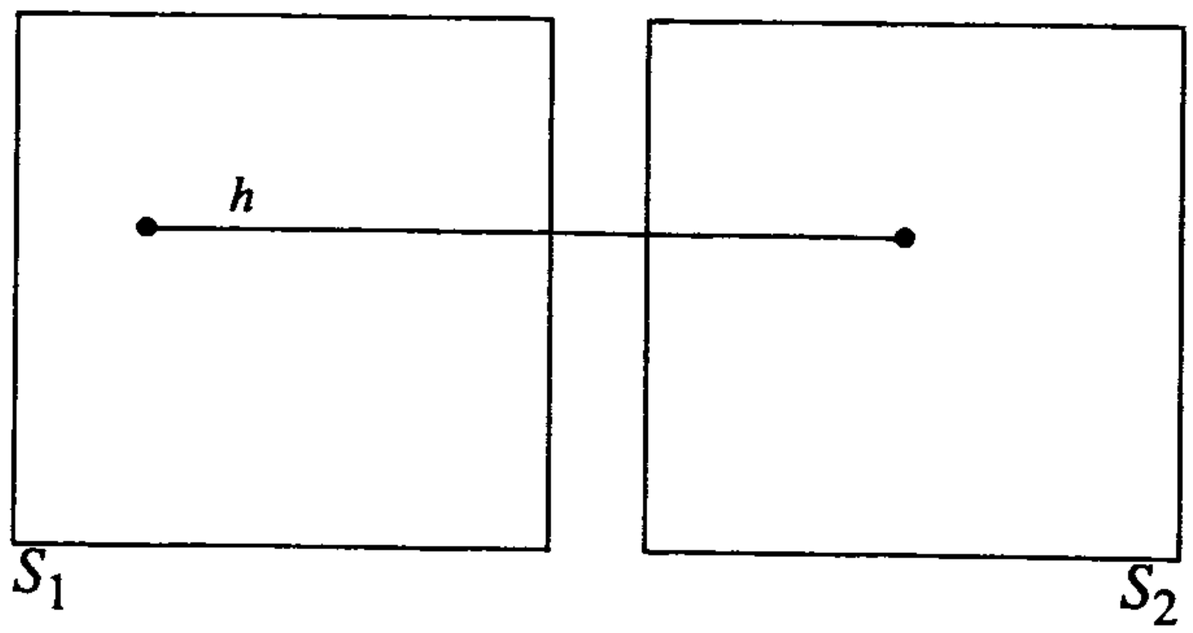
1. Beide Endpunkte von h liegen in S_1.



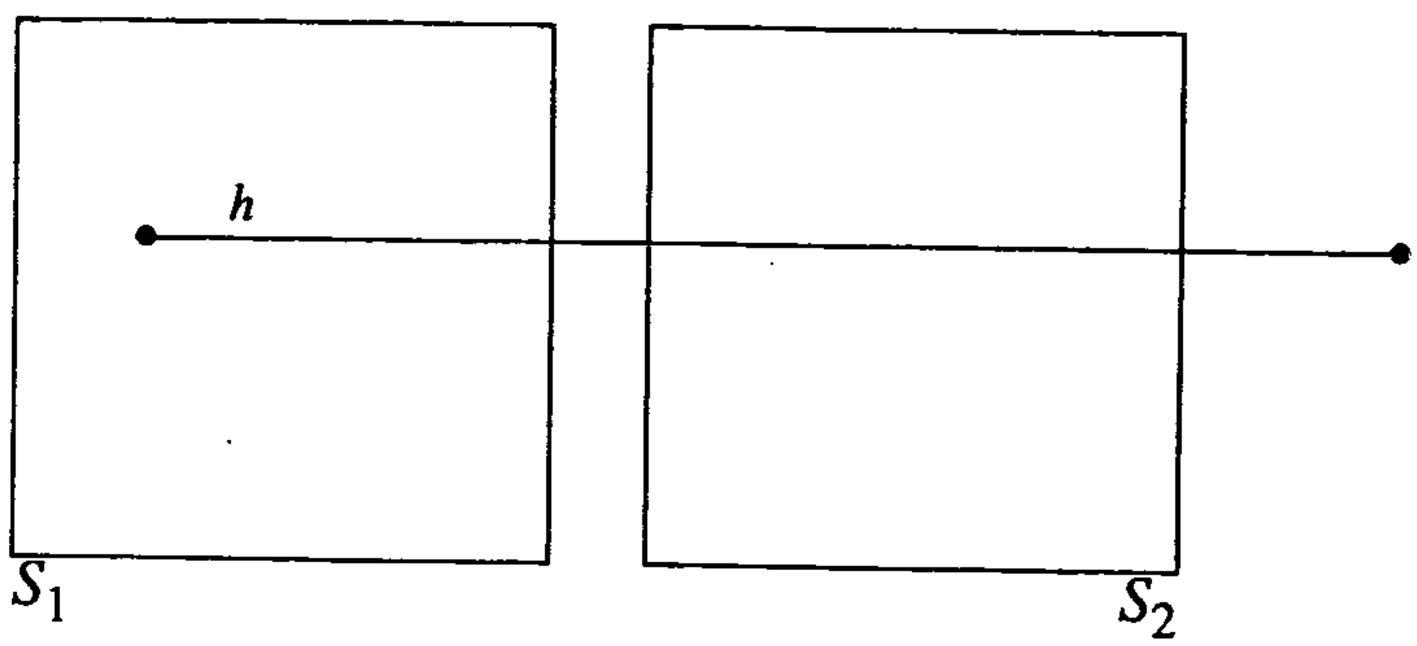
2. Der rechte Endpunkt von h liegt in S_1.



3. Der linke Endpunkt liegt in S_1 und der rechte Endpunkt liegt in S_2.



4. Der linke Endpunkt liegt in S_1 und der rechte Endpunkt liegt nicht in S_2.



Gleiches gilt umgekehrt für die Endpunkte von Segmenten in S_2.

(c)

Das Segmentschnitt-Problem kann für n Segmente mit k sich schneidenden Paaren in $O(n \log n + k)$ Zeit gelöst werden. Dazu wird $O(n)$ Speicherplatz gebraucht.

Aufgabe 6

Die Grundidee besteht darin, die Liste sukzessive zu durchlaufen und dabei die *succ*-Felder der Elemente vom Nachfolger auf den Vorgänger umzubiegen. Dazu verwenden wir drei Zeiger: x zeigt in jedem Schritt auf den Vorgänger des aktuellen Elementes, y auf das aktuelle Element, und z auf seinen Nachfolger. Der Zeiger z ist nötig, um die Restliste nach der Manipulation von $y\uparrow.succ$ nicht zu verlieren.

Im Algorithmus positionieren wir zunächst x, y und z direkt hintereinander. In jeder Iteration leiten wir $y\uparrow.succ$ auf seinen Vorgänger, also x, um. Dann bewegen wir alle drei Zeiger ein Element weiter. Zum Schluss korrigieren wir noch das *succ*-Feld des ersten und letzten logischen Elementes sowie das Kopfelement.

```
procedure Reverse(head: listelem);
var x, y, z: ↑listelem;
begin
  x := head↑.succ;
  y := x↑.succ;
  do
     z := y↑.succ;
     y↑.succ := x;
     x := y;
     y := z;
  until y↑.succ = nil;
  y↑.succ := x;
  head↑.succ↑.succ := nil;          (* succ-Feld des nun letzten Elementes auf nil setzen *)
  head↑.succ := y;                  (* ehemals letztes ist nun erstes logisches Element *)
end Reverse;
```