



FernUniversität in Hagen

**Lösungsvorschläge  
zur Hauptklausur  
1661 „Datenstrukturen I“**

**19.08.2017**

## Aufgabe 1 Labyrinth

(a)

```

algebra Labyrinth
sorts labyrinth, min4, pos
ops   create           : min4 × min4   → labyrinth
        setWall          : labyrinth × pos   → labyrinth
        setExit          : labyrinth × pos   → labyrinth
        isReachable      : labyrinth × pos   → bool
        setPerson        : labyrinth × pos   → labyrinth
        up                : labyrinth       → labyrinth
        down              : labyrinth       → labyrinth
        left              : labyrinth       → labyrinth
        right             : labyrinth       → labyrinth

```

(b)

Die Sorte *min4* verwenden wir ausschließlich beim *create* Operator. Diese Sorte stellt Zahlen dar, die nicht kleiner sind als 4. Die Menge ist wie folgt definiert:

$$\text{min4} = \{k \in \mathbb{Z} \mid k > 3\}$$

Eine Position ist ein Paar von Zahlen, die jeweils größer sind als 0:

$$\text{pos} = \{(x, y) \in \mathbb{Z}^2 \mid x > 0 \wedge y > 0\}$$

Das Labyrinth stellen wir durch eine Menge von Paaren bestehend aus Positionen und der jeweiligen Verwendung des Feldes dar. Wir stellen sicher, dass die Positionen ein Rechteck ohne Lücken bilden. Ein Feld kann entweder unbelegt sein, eine Mauer darstellen, der Ausgang sein oder ein Feld mit einer Person darstellen. Der Ausgang muss auf einem Randfeld des Labyrinths liegen. Der Ausgang und die Person dürfen höchstens ein einziges Mal auftreten.

Wir führen zunächst eine Hilfsmenge *FieldType* ein, die die möglichen Belegungen eines Felds darstellt.

$$\text{FieldType} = \{\text{Empty}, \text{Wall}, \text{Exit}, \text{Person}\}$$

Eine weitere Menge stellt alle möglichen Felder dar.

$$\text{Field} = \text{pos} \times \text{FieldType}$$

Ein Labyrinth lässt sich dann folgendermaßen beschreiben:

$$\begin{aligned}
 \text{labyrinth} = \{L \subset \text{Field} \mid & ((x_{\max}, y_{\max}), F) \in L \wedge \\
 & (((x, y), F_1) \in L : x > 1 \Rightarrow ((x-1, y), F_2) \in L) \wedge \\
 & (((x, y), F_1) \in L : y > 1 \Rightarrow ((x, y-1), F_2) \in L) \wedge \\
 & (x_{\max} > 3) \wedge \\
 & (y_{\max} > 3) \wedge \\
 & (((x, y), F_1) \in L \wedge ((x, y), F_2) \in L \Rightarrow F_1 = F_2) \wedge \\
 & (((x, y), \text{Exit}) \in L \Rightarrow x = 1 \vee y = 1 \vee x = x_{\max} \vee y = y_{\max}) \wedge \\
 & (|\{(x, y), \text{Exit}\} \in L| < 2) \wedge \\
 & (|\{(x, y), \text{Person}\} \in L| < 2)
 \end{aligned}$$

Dabei ist

$$x_{\max} = \max(\{x \mid ((x, y), K) \in L\}) \text{ und}$$

$$y_{\max} = \max(\{y \mid ((x, y), K) \in L\})$$

(c)

Die Funktionen lassen sich auf den Trägermengen wie folgt realisieren:

**functions**

Die Funktion *create* erzeugt einfach eine Menge von leeren Feldern:

$$create(xm, ym) = \{(x, y), Empty) \mid x \geq 1 \wedge y \geq 1 \wedge x \leq xm \wedge y \leq ym\}$$

Die Funktion *setWall* ersetzt ein leeres Feld durch eine Mauer:

$$setWall(L, (x, y)) = \begin{cases} L \setminus \{(x, y), Empty\} \cup \{(x, y), Wall\} & \text{falls } ((x, y), Empty) \in L \\ L & \text{sonst} \end{cases}$$

Die Funktion *setExit* ersetzt ein leeres Feld durch einen Ausgang, sofern die gegebene Position auf dem Rand liegt und noch kein Ausgang existiert.

$$setExit(L, (x, y)) = \begin{cases} L & \text{falls } (x \neq 1 \wedge y \neq 1 \wedge x \neq x_{max} \wedge y \neq y_{max}) \vee \\ & ((x, y), Wall) \in L \vee \\ & ((x, y), Person) \in L \vee \\ & \exists((x', y'), Exit) \in L \\ L \setminus \{(x, y), Empty\} & \\ \cup \{(x, y), Exit\} & \text{sonst} \end{cases}$$

mit

$$x_{max} = \max(\{x \mid ((x, y), K) \in L\}) \text{ und}$$

$$y_{max} = \max(\{y \mid ((x, y), K) \in L\})$$

Ein Feld ist sicher vom Ausgang zu erreichen, wenn es selbst der Ausgang ist. Handelt es sich um eine Mauer oder liegt die Position außerhalb des Felds, so ist das Feld unerreichbar. Für andere Felder kann man rekursiv die vier umliegenden Felder auf Erreichbarkeit testen:

$$isReachable(L, (x, y)) = \begin{cases} false & \text{falls } ((x, y), K) \notin L \\ false & \text{falls } ((x, y), Wall) \in L \\ true & \text{falls } ((x, y), Exit) \in L \\ x > 1 \wedge isReachable(L, (x-1, y)) \vee & \text{sonst} \\ x < x_{max} \wedge isReachable(L, (x+1, y)) \vee \\ y > 1 \wedge isReachable(L, (x, y-1)) \vee \\ y < y_{max} \wedge isReachable(L, (x, y+1)) \end{cases}$$

Die Funktion *setPerson* ersetzt ein leeres Feld durch ein Personenfeld, sofern die Position erreichbar ist, dieses Feld leer ist und das Labyrinth nicht bereits ein Personenfeld enthält.

$$setPerson(L, (x, y)) = \begin{cases} L \setminus \{(x, y), Empty\} \cup \{(x, y), Person\} & \text{falls } ((x, y), Empty) \in L \wedge \\ & isReachable(L, (x, y)) \wedge \\ & \nexists((x', y'), Person) \in L \\ L & \text{sonst} \end{cases}$$

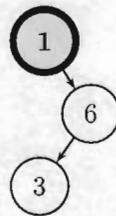
Der Operator *up* verschiebt die Position einer Person nach oben, sofern diese existiert und es sich beim Zielfeld um ein freies Feld handelt. Existiert die Person und das Zielfeld ist der Ausgang, so wird die Person aus dem Labyrinth entfernt. Ansonsten ändert sich nichts am Labyrinth.

$$up(L) = \begin{cases} L \setminus \{(x, y), Person, ((x, y+1), Empty)\} \cup \{(x, y+1), Empty, ((x, y), Person)\} & \text{falls } ((x, y), Person) \in L \wedge \\ & ((x, y+1), Empty) \in L \\ L \setminus \{(x, y), Person\} \cup \{(x, y), Empty\} & \text{falls } ((x, y), Person) \in L \\ & \wedge ((x, y+1), Exit) \in L \\ L & \text{sonst} \end{cases}$$

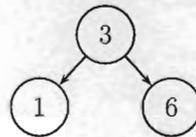
## Aufgabe 2 AVL-Baum

(a)

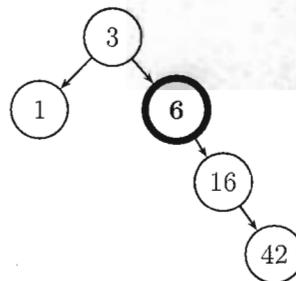
Das Einfügen von 3 führt zu einer Verletzung der Balance am markierten Knoten:



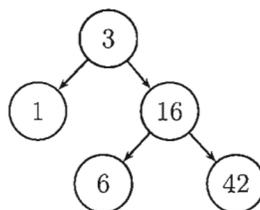
Wir führen eine Doppelrotation durch.  
Die Rotation führt zu:



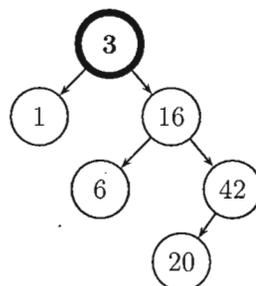
Das Einfügen von 42 führt zu einer Verletzung der Balance am markierten Knoten:



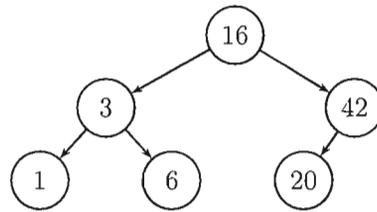
Wir führen eine Linksrotation durch.  
Die Rotation führt zu:



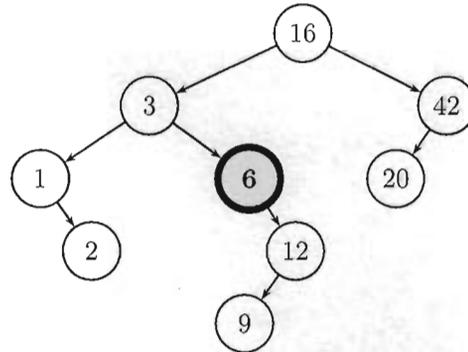
Das Einfügen von 20 führt zu einer Verletzung der Balance am markierten Knoten:



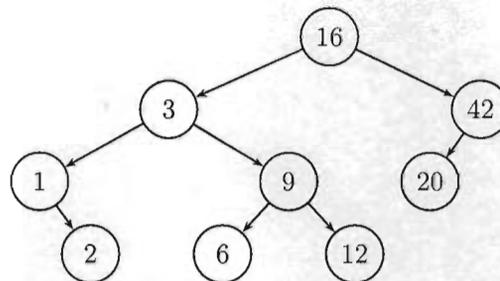
Wir führen eine Linksrotation durch.  
Die Rotation führt zu:



Das Einfügen von 9 führt zu einer Verletzung der Balance am markierten Knoten:

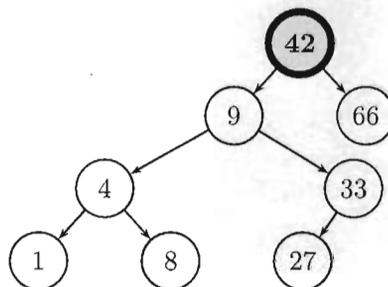


Wir führen eine Doppelrotation durch.  
Die Rotation führt zu:

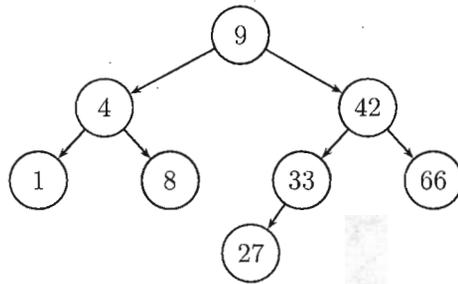


(b)

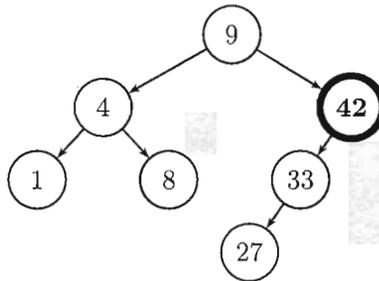
Löschen des Werts : 77  
Balance am markieren Knoten verletzt:



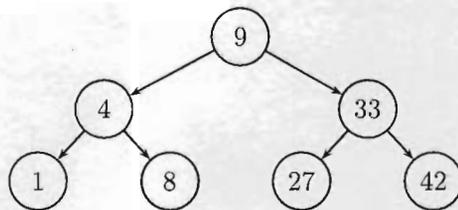
Wir führen eine Rechtsrotation durch.  
 Baum nach Löschen von 77:



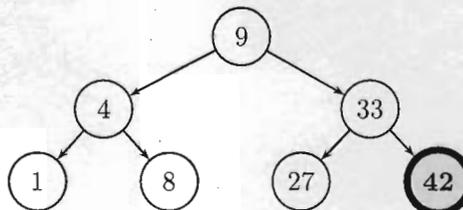
Löschen des Werts : 66  
 Balance am markieren Knoten verletzt:



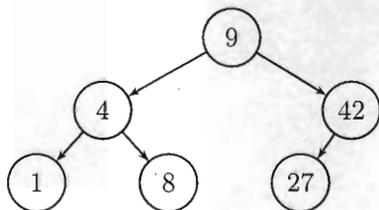
Wir führen eine Rechtsrotation durch.  
 Baum nach Löschen von 66:



Löschen des Werts : 33  
 Wir tauschen die zu löschende 33 mit dem Inhalt des markierten Knotens und löschen dann diesen.



Baum nach Löschen von 33:



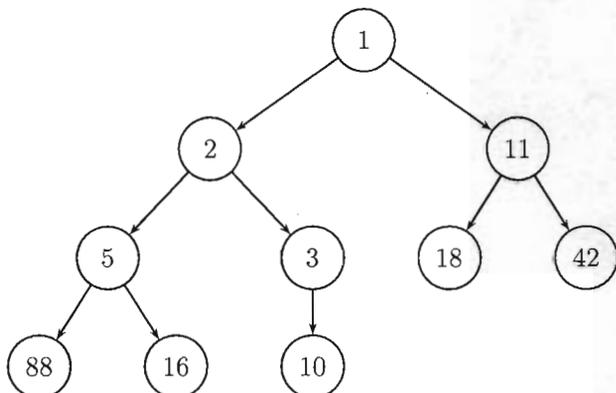
### Aufgabe 3 Sortieren

(a)

Heapsort wird gestartet

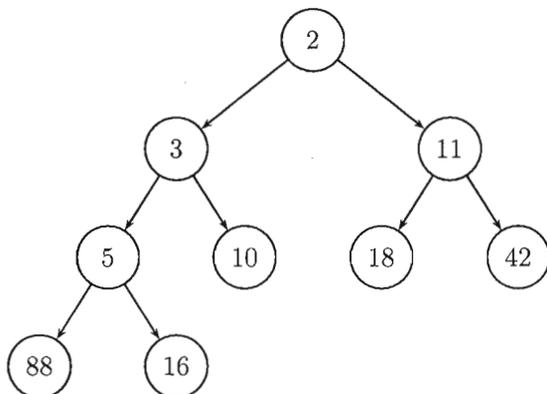
Arrayeinbettung : | 5 | 3 | 42 | 16 | 1 | 18 | 11 | 88 | 2 | 10 |

Initialer Heap erzeugt:



Arrayeinbettung : | 1 | 2 | 11 | 5 | 3 | 18 | 42 | 88 | 16 | 10 |

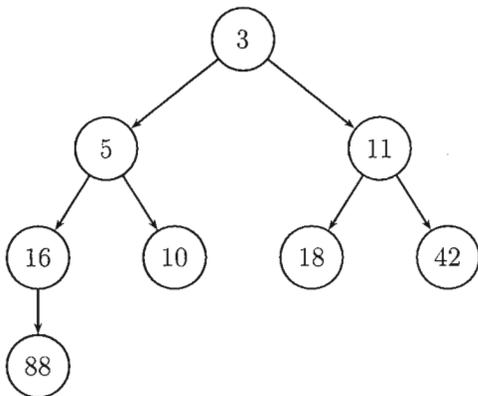
Element 1 verarbeitet:



Sortierte Folge: 1

Arrayeinbettung : | 2 | 3 | 11 | 5 | 10 | 18 | 42 | 88 | 16 || 1 |

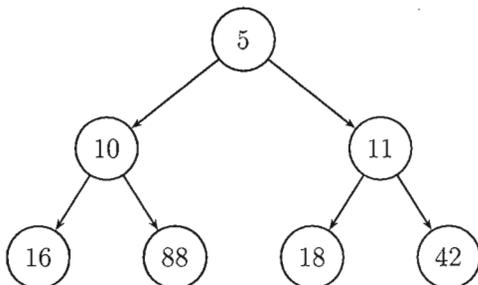
Element 2 verarbeitet:



Sortierte Folge: 2, 1

Arrayeinbettung : | 3 | 5 | 11 | 16 | 10 | 18 | 42 | 88 || 2 | 1 |

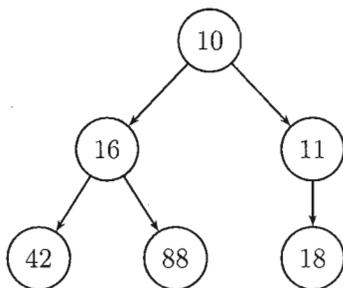
Element 3 verarbeitet:



Sortierte Folge: 3, 2, 1

Arrayeinbettung : | 5 | 10 | 11 | 16 | 88 | 18 | 42 || 3 | 2 | 1 |

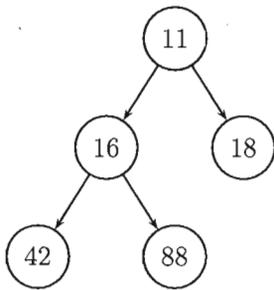
Element 5 verarbeitet:



Sortierte Folge: 5, 3, 2, 1

Arrayeinbettung : | 10 | 16 | 11 | 42 | 88 | 18 || 5 | 3 | 2 | 1 |

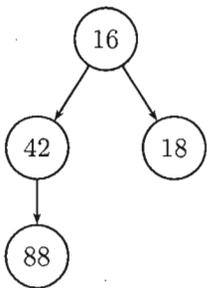
Element 10 verarbeitet:



Sortierte Folge: 10, 5, 3, 2, 1

Arrayeinbettung : | 11 | 16 | 18 | 42 | 88 || 10 | 5 | 3 | 2 | 1 |

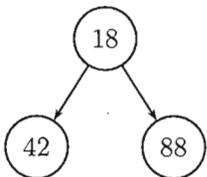
Element 11 verarbeitet:



Sortierte Folge: 11, 10, 5, 3, 2, 1

Arrayeinbettung : | 16 | 42 | 18 | 88 || 11 | 10 | 5 | 3 | 2 | 1 |

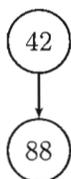
Element 16 verarbeitet:



Sortierte Folge: 16, 11, 10, 5, 3, 2, 1

Arrayeinbettung : | 18 | 42 || 88 || 16 | 11 | 10 | 5 | 3 | 2 | 1 |

Element 18 verarbeitet:



Sortierte Folge: 18, 16, 11, 10, 5, 3, 2, 1

Arrayeinbettung : | 42 | 88 || 18 | 16 | 11 | 10 | 5 | 3 | 2 | 1 |

Element 42 verarbeitet:



Sortierte Folge: 42, 18, 16, 11, 10, 5, 3, 2, 1

Arrayeinbettung : | 88 || 42 | 18 | 16 | 11 | 10 | 5 | 3 | 2 | 1 |

Element 88 verarbeitet:

Sortierte Folge: 88, 42, 18, 16, 11, 10, 5, 3, 2, 1

Arrayeinbettung : || 88 | 42 | 18 | 16 | 11 | 10 | 5 | 3 | 2 | 1 |

(b)

Um Strings korrekt zu sortieren, muss *RadixSort* deren Buchstaben von hinten nach vorn verarbeiten. Der Algorithmus beginnt an der letzten Position des längsten Strings. Es genügt nicht, an der letzten Position des kürzesten Strings anzufangen. Bei einem solchen Vorgehen wäre das Ergebnis für die Folge „aaz aa aaa“ die unveränderte Folge, was offensichtlich falsch ist. Strings, die an der aktuell zu verarbeitenden Position noch nicht existieren, werden in den Behälter  $B_0$  eingefügt.

Initiale Verteilung (Phase 0):

$$B_0 = \{\text{Oracle, Neo, Trinity, Morpheus, Apoc, Niobe, Tank, Smith, Switch}\}$$

$$B_r = \{\text{Nerowinger}\}$$

Phase: 1

$$B_0 = \{\text{Oracle, Neo, Trinity, Morpheus, Apoc, Niobe, Tank, Smith, Switch}\}$$

$$B_e = \{\text{Nerowinger}\}$$

Phase: 2

$$B_0 = \{\text{Oracle, Neo, Trinity, Apoc, Niobe, Tank, Smith, Switch}\}$$

$$B_g = \{\text{Nerowinger}\}$$

$$B_s = \{\text{Morpheus}\}$$

Phase: 3

$$B_{\emptyset} = \{\text{Oracle, Neo, Apoc, Niobe, Tank, Smith, Switch}\}$$
$$B_n = \{\text{Nerowinger}\}$$
$$B_u = \{\text{Morpheus}\}$$
$$B_y = \{\text{Trinity}\}$$

Phase: 4

$$B_{\emptyset} = \{\text{Neo, Apoc, Niobe, Tank, Smith}\}$$
$$B_e = \{\text{Oracle, Morpheus}\}$$
$$B_h = \{\text{Switch}\}$$
$$B_i = \{\text{Nerowinger}\}$$
$$B_t = \{\text{Trinity}\}$$

Phase: 5

$$B_{\emptyset} = \{\text{Neo, Apoc, Tank}\}$$
$$B_c = \{\text{Switch}\}$$
$$B_e = \{\text{Niobe}\}$$
$$B_h = \{\text{Smith, Morpheus}\}$$
$$B_i = \{\text{Trinity}\}$$
$$B_l = \{\text{Oracle}\}$$
$$B_w = \{\text{Nerowinger}\}$$

Phase: 6

$$B_{\emptyset} = \{\text{Neo}\}$$
$$B_b = \{\text{Niobe}\}$$
$$B_c = \{\text{Apoc, Oracle}\}$$
$$B_k = \{\text{Tank}\}$$
$$B_n = \{\text{Trinity}\}$$
$$B_o = \{\text{Nerowinger}\}$$
$$B_p = \{\text{Morpheus}\}$$
$$B_t = \{\text{Switch, Smith}\}$$

Phase: 7

$$B_a = \{\text{Oracle}\}$$
$$B_i = \{\text{Trinity, Switch, Smith}\}$$
$$B_n = \{\text{Tank}\}$$
$$B_o = \{\text{Neo, Niobe, Apoc}\}$$
$$B_r = \{\text{Nerowinger, Morpheus}\}$$

Phase: 8

$$B_a = \{\text{Tank}\}$$
$$B_e = \{\text{Neo, Nerowinger}\}$$
$$B_i = \{\text{Niobe}\}$$
$$B_m = \{\text{Smith}\}$$
$$B_o = \{\text{Morpheus}\}$$
$$B_p = \{\text{Apoc}\}$$
$$B_r = \{\text{Oracle, Trinity}\}$$
$$B_w = \{\text{Switch}\}$$

Phase: 9

$B_a = \{Apoc\}$

$B_m = \{Morpheus\}$

$B_n = \{Neo, Nerowinger, Niobe\}$

$B_o = \{Oracle\}$

$B_s = \{Smith, Switch\}$

$B_t = \{Tank, Trinity\}$

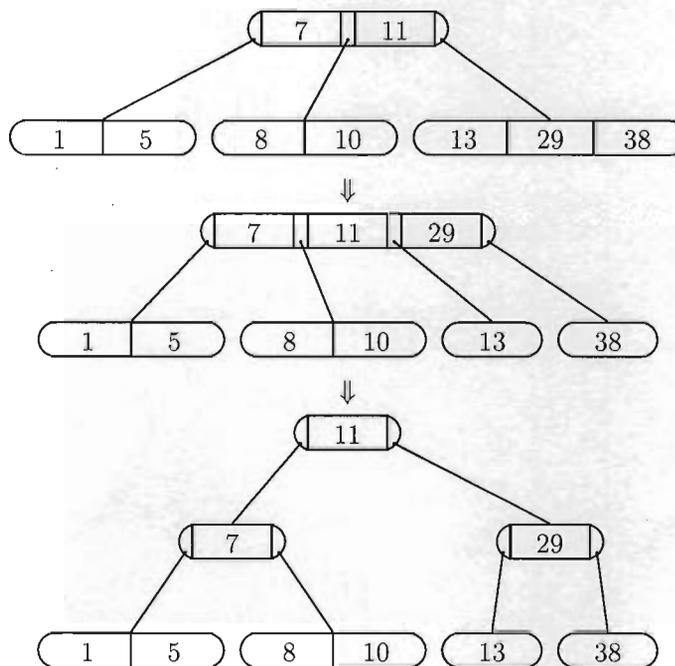
Die sortierte Folge lautet:

Apoc Morpheus Neo Nerowinger Niobe Oracle Smith Switch Tank Trinity

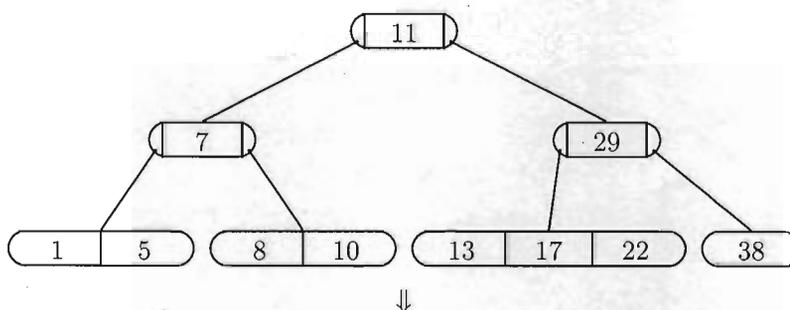
### Aufgabe 4 B-Bäume

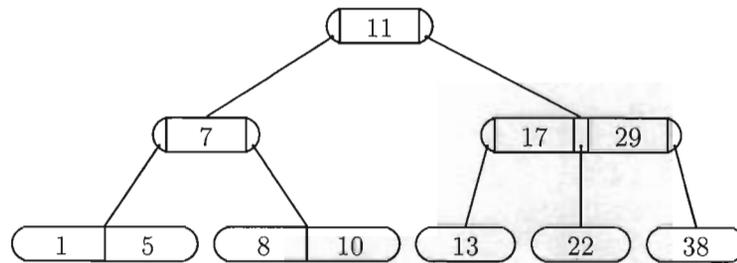
(a)

Nach dem Einfügen des Elements 29 sind zwei Splits nacheinander erforderlich.

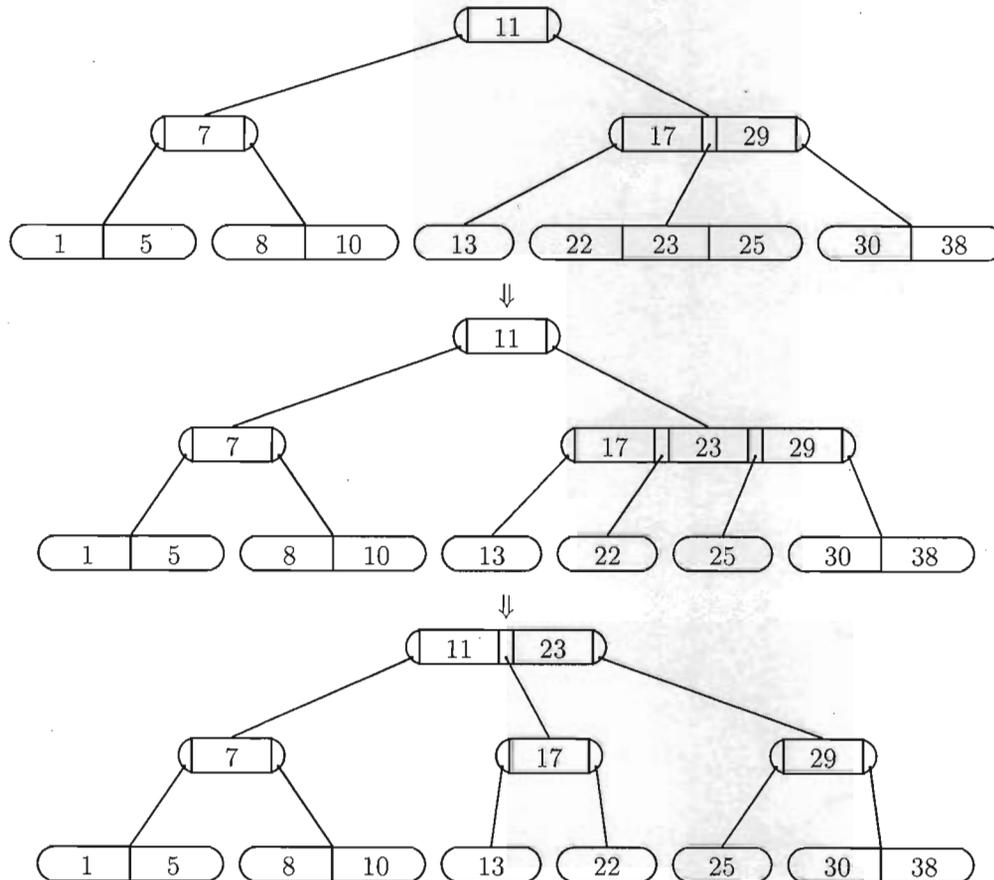


Nach dem Einfügen des Elements 22 ist ein Split erforderlich.

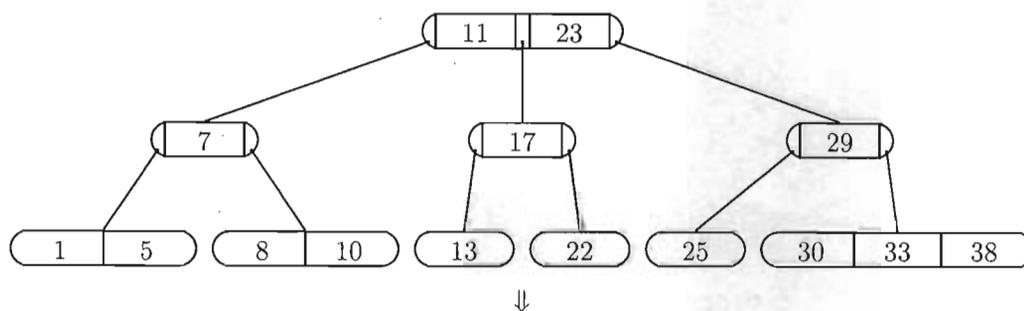


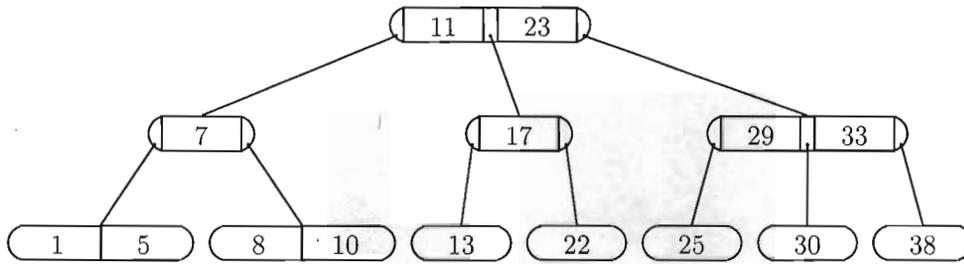


Nach dem Einfügen des Elements 23 sind zwei Splits nacheinander erforderlich.



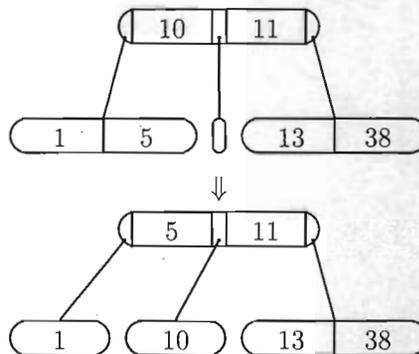
Nach dem Einfügen des Elements 33 ist ein Split erforderlich.



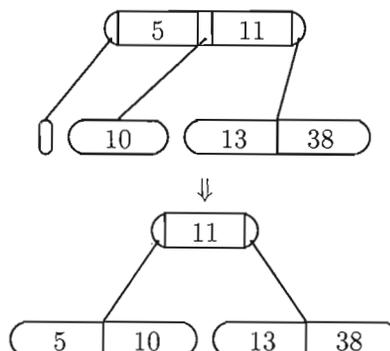


(b)

Nach dem Löschen des Elements 7 ist eine Balance-Operation erforderlich.



Nach dem Löschen des Elements 1 ist eine Merge-Operation erforderlich.



### Aufgabe 5 Deckblatt

Hier erhalten Sie den Punkt, wenn Sie beide Klausurdeckblätter korrekt und vollständig ausgefüllt haben, also Namen, Matrikelnummer und Adresse korrekt eingetragen und genau diejenigen Aufgaben markiert haben, die Sie auch bearbeitet haben.