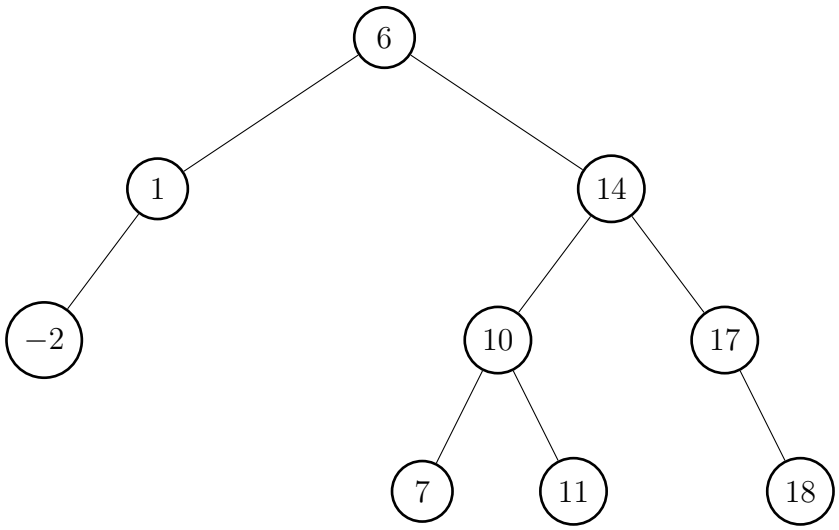
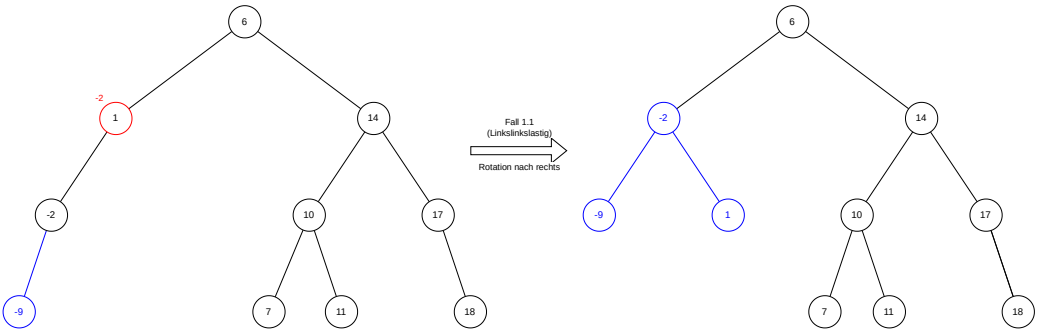
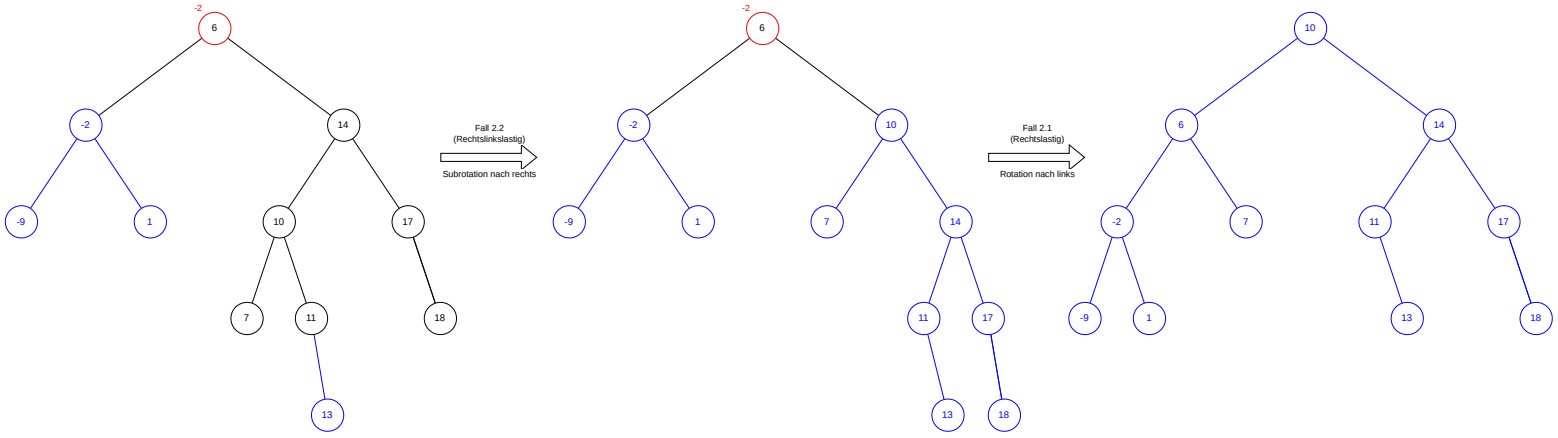
Aufgabe 1. Gegeben ist folgender **AVL-Baum**:

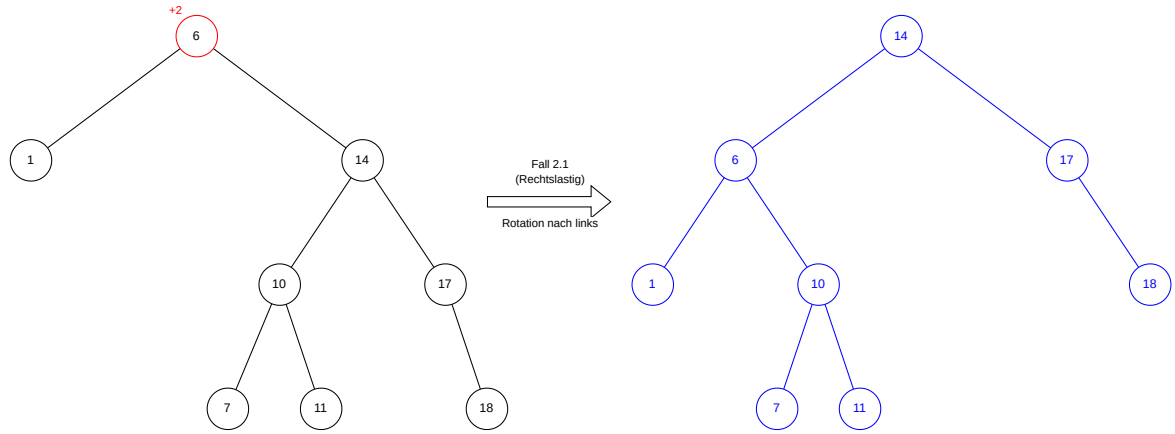


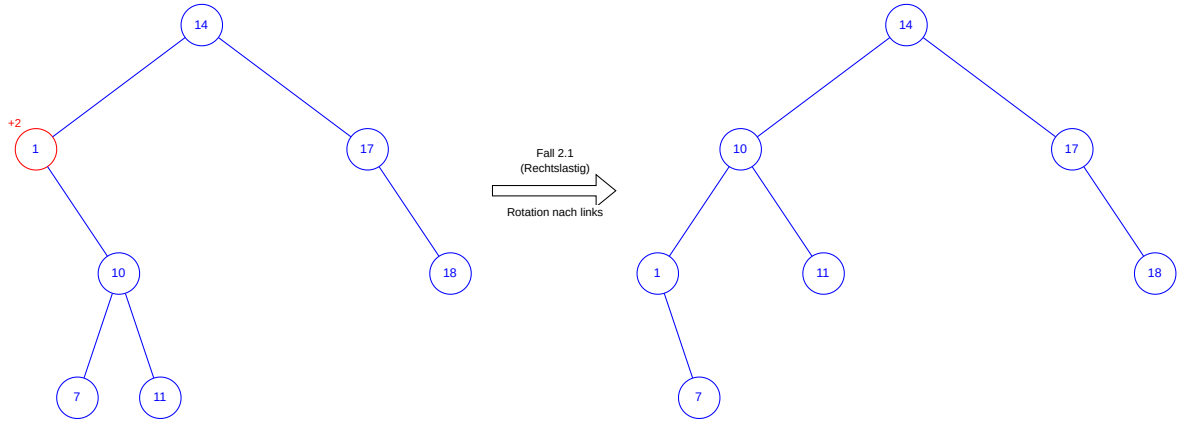
(a) Fügen Sie zuerst den Schlüssel -9 und dann 13 in den oben gegebenen AVL-Baum ein. Falls notwendig, so rebalancieren Sie den Baum nach jedem Einfügen mit geeigneten Rotationsoperationen (siehe Foliensatz „Suchbäume“), um wieder einen gültigen AVL-Baum zu erhalten. Markieren Sie dabei den unbalancierten Knoten mit maximaler Tiefe.





(b) Löschen Sie aus dem ursprünglichen in Aufgabe 1 gegebenen **AVL-Baum** die Schlüssel -2 und 6 in dieser Reihenfolge. Falls notwendig, so rebalancieren Sie den Baum nach jedem Löschvorgang mit geeigneten Rotationsoperationen (siehe Foliensatz „Suchbäume“), um wieder einen gültigen AVL-Baum zu erhalten. Markieren Sie dabei den unbalancierten Knoten mit maximaler Tiefe.

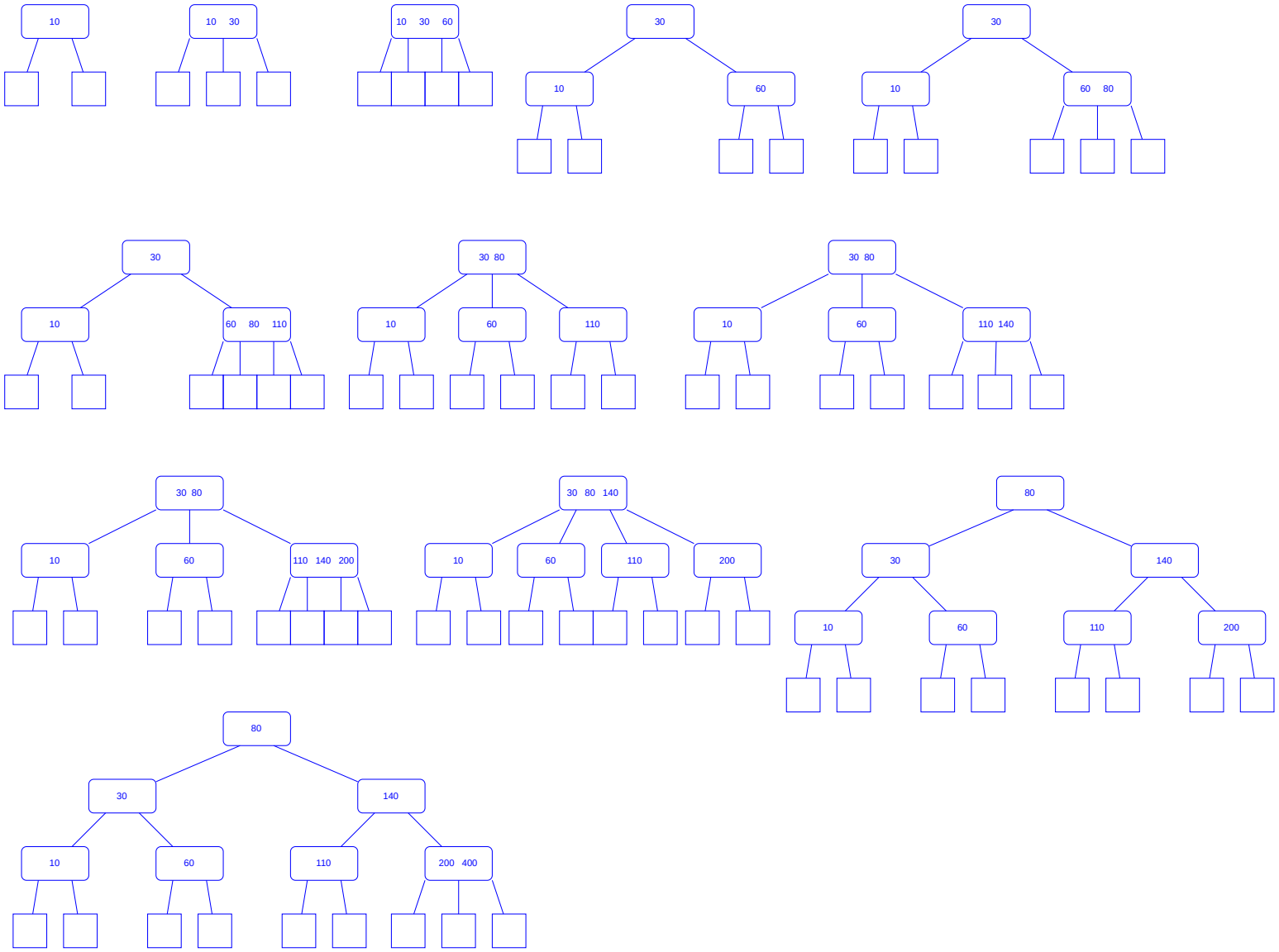




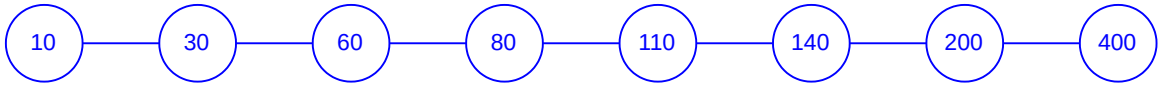
Aufgabe 2. Gegeben sei die folgende Folge von Elementen:

$$\langle 10, 30, 60, 80, 110, 140, 200, 400 \rangle$$

(a) Fügen Sie die Elemente in dieser Reihenfolge in einen anfangs leeren B-Baum der Ordnung 3 ein. Zeichnen Sie den B-Baum jeweils vor und nach jeder Reorganisationsmaßnahme und geben Sie den endgültigen B-Baum an.

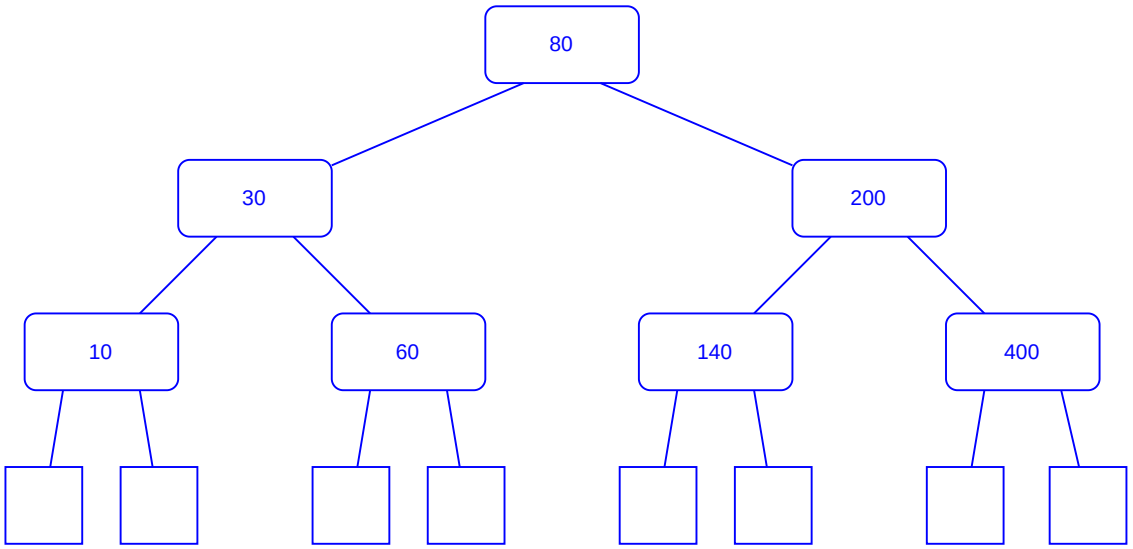


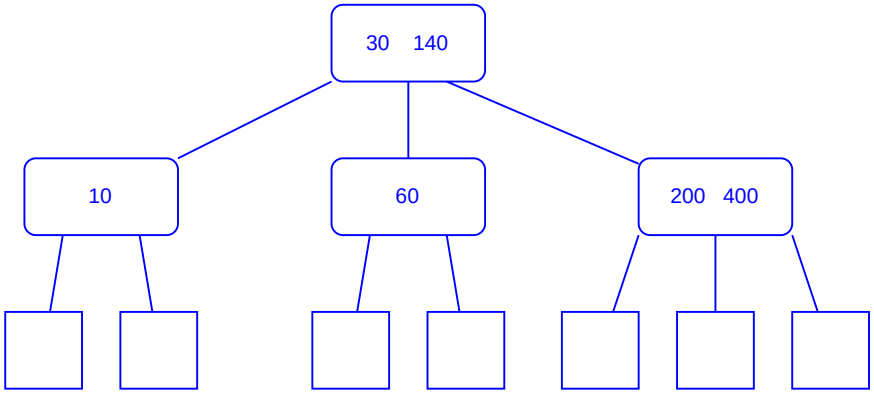
(b) Fügen Sie die Folge auch in einen natürlichen binären Suchbaum ein und vergleichen Sie das Resultat mit dem B-Baum aus Unteraufgabe (a).



Im Gegensatz zum Baum aus Aufgabe (a) ist der natürliche binäre Suchbaum nicht ausbalanciert sondern zu einer Liste entartet

(c) Geben Sie den B-Baum an, der durch Löschen der Schlüssel 110 und 80 (in dieser Reihenfolge) aus dem B-Baum von Unteraufgabe (a) entsteht.





Aufgabe 3. Geben Sie einen Algorithmus in detailliertem Pseudocode an, der folgendes Problem löst:

Die Eingabe ist ein Array A. Falls A einer gültigen **postorder** Traversierung eines **binären Suchbaums** entspricht, soll Ihr Algorithmus einen solchen binären Suchbaum erstellen und dessen Wurzelknoten zurückgeben. Andernfalls soll der Algorithmus als Rückgabewert *false* zurückgeben. Sie können annehmen, dass alle Elemente in A paarweise unterschiedliche natürliche Zahlen sind. Ihnen stehen dafür folgende Operationen zur Verfügung:

- NEWNODE(x): diese Methode erstellt einen neuen Knoten mit Wert x und leerem linken und rechten Kind und gibt diesen neuen Knoten zurück.
- v.left $\leftarrow v_\ell$: dies weist dem linken Kind von dem Knoten v den Knoten v_ℓ zu. Das heißt, danach ist v_ℓ das linke Kind von v.
- v.right $\leftarrow v_r$: dies weist dem rechten Kind von dem Knoten v den Knoten v_r zu. Das heißt, danach ist v_r das rechte Kind von v.

Hinweis 1: Es gibt maximal einen binären Suchbaum, dessen postorder Traversierung A entspricht. Wieso?
Hinweis 2: Versuchen Sie den Algorithmus als rekursive Funktion zu beschreiben.

```
CreateBinaryTree(A):
    I = sorted(A)  # Sort A, gets in-order traversal of the tree (because the in-order
    traversal of a binary tree is always the key sorted ascending)
    tree = BuildTree(A, I)

    if IsValidTree(tree, -oo, oo):
        return tree
    else:
        return False


BuildTree(A, I):
    if len(A) < 1:
        return null

    root = NewNode(A.pop())
    separator = I.index(root.key)  # Get the index of the root keyval in the in-order
    list
    right_subtree = A[separator:]  # The elements of the right subtree are all elements
    with index >= separator (if separator > len(A) right_subtree = [])
    left_subtree = A[:separator]   # All other elements compose the left subtree
    right_I = I[separator+1:]      # Align the index list that the offsets match again.
    Exclude the root.key element
    left_I = I[:separator]         # All the other elements belong to the left offset
    list

    root.right = BuildTree(right_subtree, right_I)
    root.left = BuildTree(left_subtree, left_I)

    return root


IsValidTree(tree, lowVal, upVal):
    if tree == null:
        return True

    if tree.key < lowVal or tree.key > upVal:
        return False

    return IsValidTree(tree.left, lowVal, tree.key) and IsValidTree(tree.right, tree.key
    , upVal)
```

Aufgabe 4. Gegeben ist folgende Hashtabelle der Größe $m = 11$:

Position	0	1	2	3	4	5	6	7	8	9	10
Schlüssel	44			3			17	7	30	20	

Führen Sie folgende Schritte in der vorgegebenen Reihenfolge aus:

1. Füge 13 ein
2. Füge 6 ein
3. Lösche 7
4. Suche 6
5. Füge 28 ein

Tun Sie dies für **jede** der folgenden Varianten zur Kollisionsbehandlung. Beschreiben Sie die einzelnen Schritte und stellen Sie die finale Belegung dar.

(a) Verkettung der Überläufer mit $h(k) = k \bmod m$.

```
Insert 13:
- 13 % 11 = 2
- Pos. 2 == null => insert element

Insert 6:
- 6 % 11 = 6
- Pos. 6 != null => append to end of list

Delete 7:
- 7 % 11 = 7
- Perform linear search on pos. 7 => find it a the beginning => remove
    element from list (results in empty list (=null))

Search 6:
- 6 % 11 = 6
- Perform linear search on pos. 6 => find element at end of list

Insert 28:
- 28 % 11 = 6
- Pos. 6 != null => append to end of list
```

```
 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10
---+---+---+---+---+---+---+---+---+---+---
 44|   | 13| 3 |   |   | * | 7 | 30| 20|
*: 17 - 6 - 28
```

(b) Lineares Sondieren mit $h(k,i) = (h'(k) + i) \bmod m$ und $h'(k) = k \bmod m$.

```
Insert 13:
 - 13 % 11 = 2
 - Pos. 2 == null => insert element, set flag "used"

Insert 6:
 - 6 % 11 = 6
 - Pos. 6 != null => 6+1=7
 - Pos. 7 != null => 6+2=8
 - Pos. 8 != null => 6+3=9
 - Pos. 9 != null => 6+4=10
 - Pos. 10 == null => insert element, set flag "used"

Delete 7:
 - 7 % 11 = 7
 - Element on pos. 7 == 7 => remove element

Search 6:
 - 6 % 11 = 6
 - Element on pos. 6 != 6 => 6+1=7
 - Element on pos. 7 == null, but "used" flag is set => 6+2=8
 - Element on pos. 8 != 6 => 6+3=9
 - Element on pos. 9 != 6 => 6+4=10
 - Element on pos. 10 == 6 => found

Insert 28:
 - 28 % 11 = 6
 - Pos. 6 != null => 6+1=7
 - Pos. 7 == null => insert element, set flag "used"
```

0	1	2	3	4	5	6	7	8	9	10
44		13	3			17	28	30	20	6
u	f	u	u	f	f	u	u	u	u	u

(c) Quadratisches Sondieren mit $h(k,i) = (h'(k) + \frac{1}{2}i + \frac{1}{2}i^2) \bmod m$ und $h'(k) = k \bmod m$.

```
Insert 13:
 - 13 % 11 = 2
 - Pos. 2 == null => insert element, set flag "used"

Insert 6:
 - 6 % 11 = 6
 - Pos. 6 != null => 6+(1+1)/2=7
 - Pos. 7 != null => 6+(2+4)/2=9
 - Pos. 9 != null => 6+(3+9)/2=1 (mod 11)
 - Pos. 1 == null => insert element, set flag "used"

Delete 7:
 - 7 % 11 = 7
 - Element on pos. 7 == 7 => remove element

Search 6:
 - 6 % 11 = 6
 - Element on pos. 6 != 6 => 6+(1+1)/2=7
 - Element on pos. 7 == null, but "used" flag is set => 6+(2+4)/2=9
 - Element on pos. 9 != 6 => 6+3=9
 - Element on pos. 9 != 6 => 6+(3+9)/2=1 (mod 11)
 - Element on pos. 1 == 6 => found

Insert 28:
 - 28 % 11 = 6
 - Pos. 6 != null => 6+(1+1)/2=7
 - Pos. 7 == null => insert element, set flag "used"
```

0	1	2	3	4	5	6	7	8	9	10
44	6	13	3			17	28	30	20	
u	u	u	u	f	f	u	u	u	u	f

Für das lineare und quadratische Sondieren gilt $i = 0, 1, \ldots, m-1$. Nehmen Sie beim linearen und quadratischen Sondieren an, dass zu Beginn alle leeren Felder in der Hashtabelle mit dem Flag „frei“ markiert sind.

Aufgabe 5. Gegeben ist folgende Hashtabelle der Größe $m = 7$.

Position	0	1	2	3	4	5	6
Schlüssel	42		2	3		12	

Wir verwenden Double Hashing mit $h_1(k) = k \bmod m$ und $h_2(k) = (k \bmod 5) + 1$.

(a) Fügen Sie 66 in die obige Tabelle ein. Geben Sie alle Zwischenschritte an.

```
Insert 66:
 - 66 % 7 = 3
 - Pos. 3 != null => 3+1*(66 % 5 + 1)=5
 - Pos. 5 != null => 3+2*(66 % 5 + 1)=0 (mod 7)
 - Pos. 0 != null => 3+3*(66 % 5 + 1)=2 (mod 7)
 - Pos. 2 != null => 3+4*(66 % 5 + 1)=4 (mod 7)
 - Pos. 4 == null => insert element, set flag "used"
```

```
 0 | 1 | 2 | 3 | 4 | 5 | 6
---+---+---+---+---+---+---
42|   | 2 | 3 | 66| 12|
```

```
Durchschnittliche Suchzeit:
 42|  2|  3| 66| 12
---+---+---+---+---
1 + 1 + 1 + 5 + 1 = 9
9/5 = 1.8
```

(b) Fügen Sie erneut 66 in die obige (ursprüngliche) Tabelle ein. Verwenden Sie diesmal dazu das verbesserte Einfügen nach Brent. Geben Sie alle Zwischenschritte an.

```
Insert 66:
 - 66 % 7 = 3
 - T[3].status == used
   - j1 = 3 + 66 % 5 + 1 = 5
   - j2 = 3 +  3 % 5 + 1 = 0 (mod 7)
   - T[j2].status == used => j = j1 = 5
 - T[5].status == used
   - j1 = 5 + 66 % 5 + 1 = 0
   - j2 = 5 + 12 % 5 + 1 = 1 (mod 7)
   - T[j1].status == used and T[j2].status != used =>
     - T[1] = 12
     - T[5] = 66
```

```
 0 | 1 | 2 | 3 | 4 | 5 | 6
---+---+---+---+---+---+---
42| 12| 2 | 3 |   | 66|
```

```
Durchschnittliche Suchzeit:
 42| 12|  2|  3| 66
---+---+---+---+---
1 + 2 + 1 + 1 + 2 = 7
7/5 = 1.4
```

(c) Bestimmen Sie die durchschnittliche Anzahl an Schritten für eine erfolgreiche Suche in der Tabelle die Sie in Unteraufgabe (a) nach dem Einfügen erhalten. Gehen Sie analog für die Tabelle, die Sie aus Unteraufgabe (b) erhalten, vor.

```
See (a) or (b) respectively
```