Tutorium Grundlagen: Algorithmen und Datenstrukturen

Übungsblatt Woche 11

Aufgabe 11.1
Dirty Double Hashing

Aufgabe 11.1

Für diese Aufgabe verwenden wir ein modifiziertes Double Hashing, welches beim Löschen von Elementen die abhängigen Kollisionen nicht neu hasht, sondern einfach einen “gelöscht”- Platzhalter einfügt (mit  zu markieren).
Die Größe der Hashtabelle ist $m = 11$. Die Schlüssel der Elemente sind die Elemente selbst. Verwenden Sie die folgenden Hashfunktionen:

$$h(x, i) = (h(x) + i * h'(x)) \bmod 11$$
$$h(x) = 3x \bmod 11$$
$$h'(x) = 1 + (x \bmod 10)$$

a) Unter welchen zwei Umständen kann die Find-Operation ergebnislos abbrechen?
b) Können bei dieser Vorgehensweise die Platzhalter beim Einfügen überschrieben werden?
c) Wir fügen $n > 0$ Elemente in eine anfangs leere Hashtabelle ausreichender Größe ein und löschen diese wieder. Was ist die Worst-Case Laufzeit einer folgenden FindOperation? Begründen Sie Ihre Antwort kurz.
d) Führen Sie folgende Operationen in der gegebenen Reihenfolge aus (tragen Sie auch alle überprüften Positionen ein):

$$\begin{array}{ll} \texttt{insert:} & 4, 15, 6, 10 \\ \texttt{delete:} & 4, 10 \\ \texttt{insert:} & 10, 1 \end{array}$$

Aufgabe 11.1

a) Unter welchen zwei Umständen kann die Find-Operation ergebnislos abbrechen?

b) Können bei dieser Vorgehensweise die Platzhalter beim Einfügen überschrieben werden?

c) Wir fügen $n > 0$ Elemente in eine anfangs leere Hashtabelle ausreichender Größe ein und löschen diese wieder. Was ist die Worst-Case Laufzeit einer folgenden FindOperation? Begründen Sie Ihre Antwort kurz.

Aufgabe 11.1

1. Operation: `insert`(4):

0	1	2	3	4	5	6	7	8	9	10

2. Operation: `insert`(15):

0	1	2	3	4	5	6	7	8	9	10

3. Operation: `insert`(6):

0	1	2	3	4	5	6	7	8	9	10

4. Operation: `insert`(10):

0	1	2	3	4	5	6	7	8	9	10

5. Operation: `delete`(4):

0	1	2	3	4	5	6	7	8	9	10

6. Operation: `delete`(10):

0	1	2	3	4	5	6	7	8	9	10

7. Operation: `insert`(10):

0	1	2	3	4	5	6	7	8	9	10

8. Operation: `insert`(1):

0	1	2	3	4	5	6	7	8	9	10

	4	15	6	10	1
i = 0	1	1	7	8	3
i = 1	6	7	3	9	5
i = 2	0	2	10	10	7

Aufgabe 11.2
Multiple Choice

Aufgabe 11.2 (a)

Wir erweitern den Bubblesort-Algorithmus, indem vor **jeder** Vergleichsoperation eine Funktion $\mathrm{isSorted}()$ aufgerufen wird. Diese überprüft, ob das gesamte Feld bereits sortiert ist, indem jedes Paar von benachbarten Elementen überprüft wird. Dazu wird das gesamte zu sortierende Feld einmal komplett durchlaufen. Ist das Feld sortiert, wird das modifizierte Sortierverfahren sofort beendet.

Es bezeichne f die **Worst-Case**-Laufzeit des Original-Bubblesort-Algorithmus und g die des modifizierten Bubblesort-Algorithmus. Was gilt?

☐ $f \in \Theta(g)$

☐ $f \notin \Theta(g)$, aber $f \in \mathcal{O}(g)$

☐ $f \notin \Theta(g)$ und $g \in \mathcal{O}(f)$

☐ weder $g \in \mathcal{O}(f)$ noch $f \in \mathcal{O}(g)$

Aufgabe 11.2 (b)

Kreuzen Sie in den Zeilen (1) bis (3) jeweils das stärkste passende Symbol an. D. h. wenn z.B. $\Delta = o$ (bzw. $\Delta = \Theta$) möglich ist, wählen Sie $\Delta = o$ (bzw. $\Delta = \Theta$) und nicht $\Delta = \mathcal{O}$.
Falls die Funktionen unvergleichbar sind, kreuzen Sie u. (" unvergleichbar") an.
Setzen Sie also in jeder Zeile genau ein Kreuz!

Bsp:	$n \in \Delta(n^2)$	☒ o	☐ $\mathcal{O}$	☐ ω	☐ Ω	☐ Θ	☐ $u.$
(1)	$7\log_2(n) \in \Delta(4\log_2(n^2))$	☐ o	☐ $\mathcal{O}$	☐ ω	☐ Ω	☐ Θ	☐ $u.$
(2)	$n^3 \in \Delta(n^8(n \bmod 2))$	☐ o	☐ $\mathcal{O}$	☐ ω	☐ Ω	☐ Θ	☐ $u.$
(3)	$4^n \in \Delta(2^{4n})$	☐ o	☐ $\mathcal{O}$	☐ ω	☐ Ω	☐ Θ	☐ $u.$

Aufgabe 11.2 (c)

Welche Aussagen sind wahr?

- ☐ Jeder AVL-Baum ist zugleich ein binärer Suchbaum.
- ☐ Jeder binäre Suchbaum ist zugleich ein Binärer Heap.
- ☐ Jeder Binäre Heap ist zugleich ein AVL-Baum.
- ☐ Jeder Binäre Heap ist zugleich ein binärer Suchbaum.

Aufgabe 11.2 (d) & (e)

d) Welche Operation muss in einem Binären Min-Heap beim Ausführen von decreaseKey (Verringern eines Schlüssel-Wertes) u.U. aufgerufen werden, um die Heap-Invariante wiederherzustellen?

☐ `siftUp` ☐ `siftDown` ☐ `deleteMin` ☐ `rotateLeft`

e) Die durchschnittliche Laufzeit von Algorithmen (*Average Case*) ist

☐ uninteressant, da man ausschließlich am *Worst-Case* interessiert ist.

☐ oft nur mit großem Aufwand zu berechnen.

☐ stets am besten geeignet, um den passenden Algorithmus zu wählen.

Aufgabe 11.2 (f) & (g)

f) Der MergeSort-Algorithmus

☐ ist für alle Eingaben schneller als jede gute Implementierung von InsertionSort.

☐ ist für bestimmte Eingabeklassen signifikant schneller als eine deterministische Implementierung von Quicksort.

☐ ist im Schnitt um einen in der Eingabe linearen Faktor schneller als Quicksort.

☐ sortiert eine Eingabe im *Best Case* in linearer Zeit.

g) Beim Hashing mit *linear probing*

☐ werden Elemente, deren Schlüssel auf den gleichen Wert gehasht werden, in einer Liste abgelegt.

☐ ist die Hashfunktion nicht besonders wichtig, da auf ineffiziente Listen verzichtet wird.

☐ ist das Löschen von Elementen kompliziert, da Löcher in der Hashtabelle das Auffinden von anderen Elementen verhindern können.

Aufgabe 11.2 (h)

Der PermutationSort-Algorithmus sortiert ein Feld, indem er die Elemente wiederholt umordnet und jede so entstandene Anordnung auf Sortiertheit prüft. Es werden systematisch alle möglichen Anordnungen durchprobiert. Wir gehen im Folgenden davon aus, dass PermutationSort ein Feld ohne Duplikate sortiert.

☐ Für die *Worst-Case*-Laufzeit f von PermutationSort gilt $f \in \mathcal{O}(n^4)$.

☐ Für die *Average-Case*-Laufzeit f von PermutationSort gilt $f \in \Theta(n * n!)$.

☐ Für die *Worst-Case*-Laufzeit f von PermutationSort gilt $f \in \mathcal{O}(n^n)$.

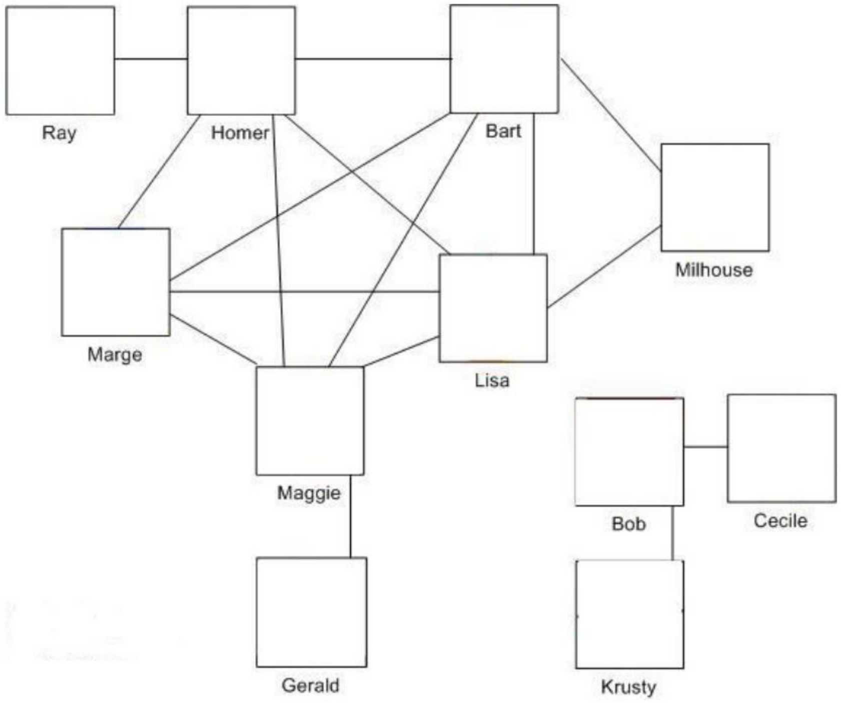
☐ Mit einer sehr geringen, positiven Wahrscheinlichkeit terminiert PermutationSort für bestimmte Eingaben niemals.

☐ PermutationSort hat eine lineare *Best-Case*-Laufzeit.

Aufgabe 11.3 Netzwerk

Aufgabe 11.3

Wir stellen uns ein soziales Netzwerk als Graph vor. In diesem Graph bilden die Personen des Netzwerks die Knoten. Sind zwei Personen befreundet, gibt es im Graphen eine ungerichtete Kante zwischen den entsprechenden Knoten. Das folgende Bild zeigt einen Ausschnitt aus einem solchen Graphen:



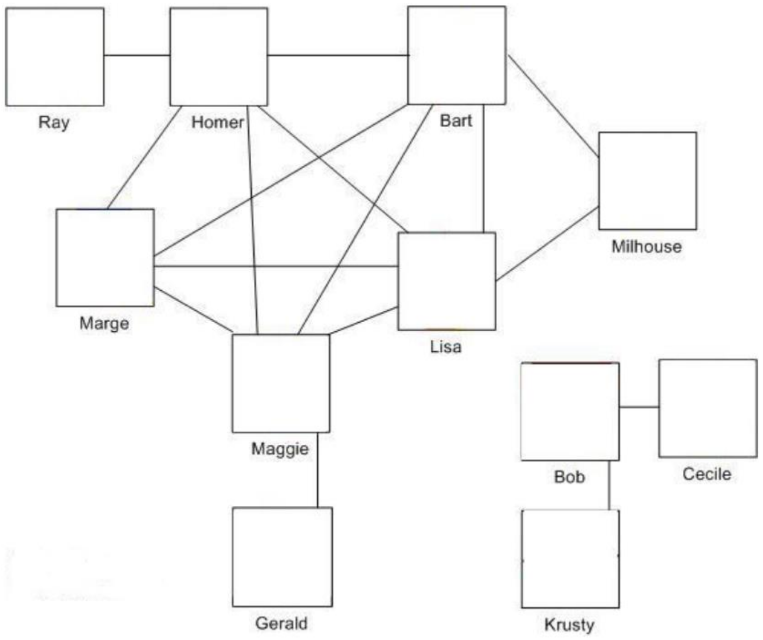
Aufgabe 11.3

a) Wie kann man mit Hilfe der Tiefensuche feststellen, ob es einen Weg zwischen zwei Personen in dem sozialen Netzwerk gibt? In diesem Fall sagen wir, dass sich diese beiden Personen kennen.

b) Überlegen Sie sich einen Algorithmus (basierend auf der Breitensuche), um festzustellen, über wie viele Personen sich zwei Personen minimal kennen. Z.B. kennt Milhouse Gerald minimal über zwei weitere Person (Lisa oder Bart und Maggie).

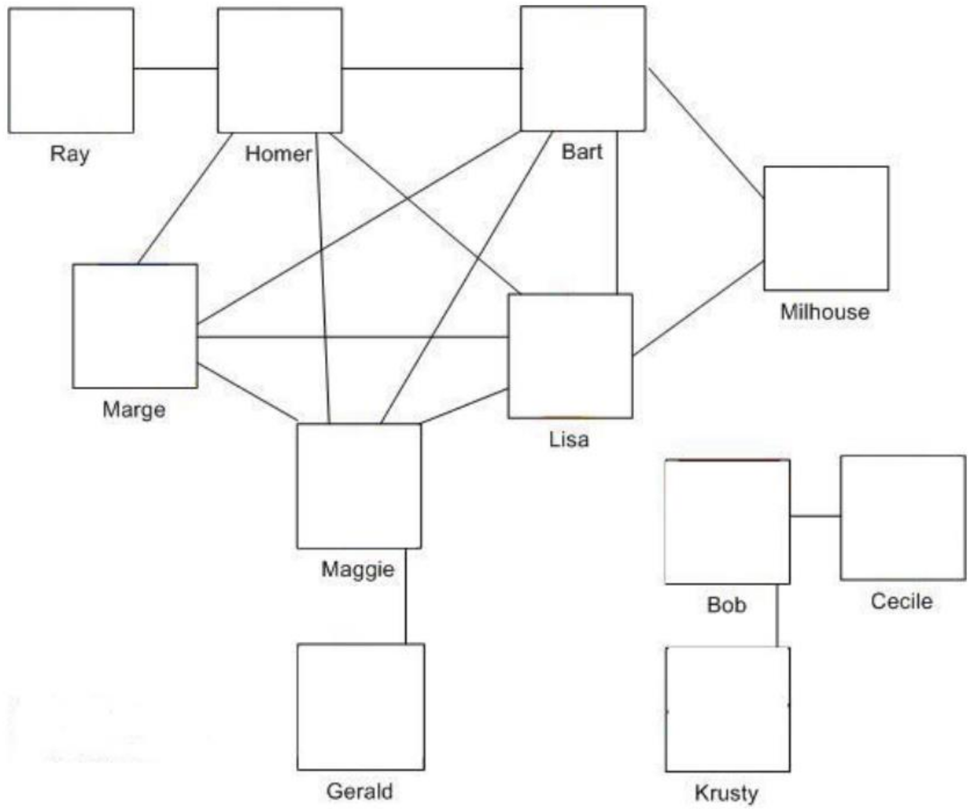
c) Ergänzen Sie Ihren Algorithmus, so dass er eine kürzeste Verbindung ausgibt.
Z.B. Milhouse - Gerald → ”Milhouse - Bart - Maggie - Gerald“

d) Überlegen Sie sich, wie man diese Aufgabe in einem gerichteten Graphen lösen könnte.
Z.B. Gerichtete Kante (Marge, Krusty): Marge kennt Krusty, dieser kennt Marge aber nicht.
Wie findet man nun heraus, dass es zwischen zwei Personen in beide Richtungen eine Verbindung gibt?



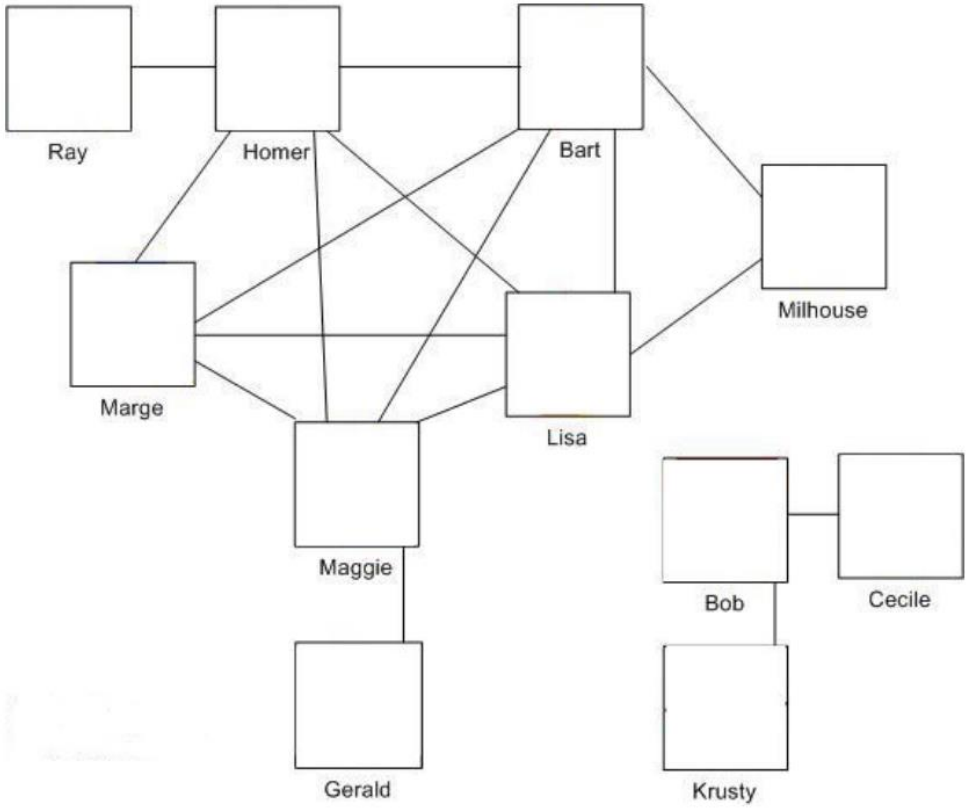
Aufgabe 11.3 (a)

Wie kann man mit Hilfe der Tiefensuche feststellen, ob es einen Weg zwischen zwei Personen in dem sozialen Netzwerk gibt? In diesem Fall sagen wir, dass sich diese beiden Personen kennen.



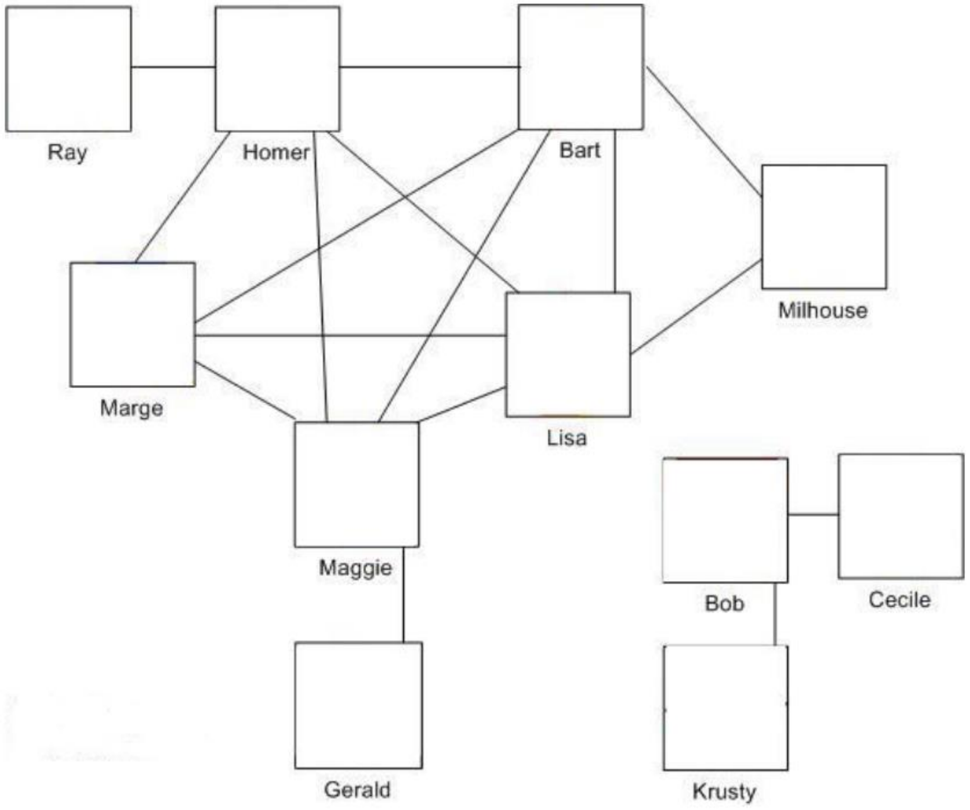
Aufgabe 11.3 (b)

Überlegen Sie sich einen Algorithmus (basierend auf der Breitensuche), um festzustellen, über wie viele Personen sich zwei Personen minimal kennen.
Z.B. kennt Milhouse Gerald minimal über zwei weitere Person (Lisa oder Bart und Maggie).



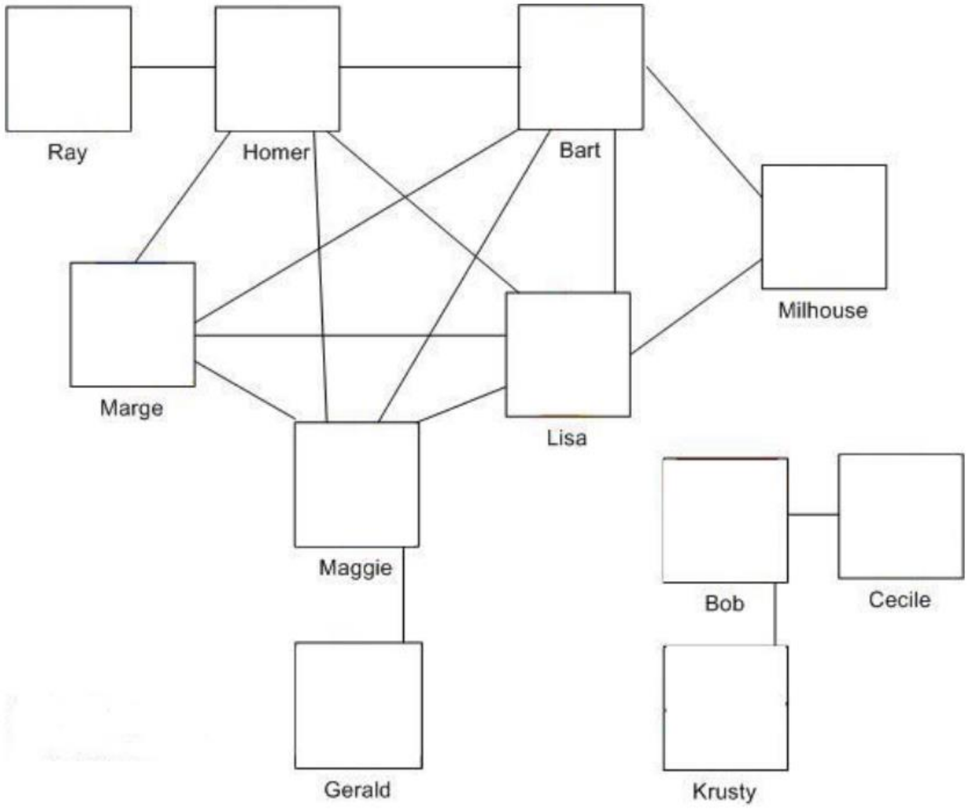
Aufgabe 11.3 (c)

Ergänzen Sie Ihren Algorithmus, so dass er eine kürzeste Verbindung ausgibt.
Z.B. Milhouse - Gerald → ”Milhouse - Bart - Maggie - Gerald“



Aufgabe 11.3 (d)

Überlegen Sie sich, wie man diese Aufgabe in einem gerichteten Graphen lösen könnte.
Z.B. Gerichtete Kante (Marge, Krusty): Marge kennt Krusty, dieser kennt Marge aber nicht.
Wie findet man nun heraus, dass es zwischen zwei Personen in beide Richtungen eine Verbindung gibt?



Aufgabe 11.4 Klausurvorbereitung: Lückencode

Aufgabe 11.4

Diese Aufgabe zeigt ein mögliche Aufgabenstellung für die Klausur. Es geht darum zu zeigen, dass Sie den Stoff aus der Vorlesung verstanden haben und selbstständig ein gegebenes Codegerüst vervollständigen können. Es ist auch möglich, dass in der Klausur unbekannter Code vervollständigt werden soll.

Vervollständigen Sie folgenden Code zum Löschen von Elementen aus der untersten Ebene eines AB-Baumes. Es soll sich jeweils der Vaterknoten darum kümmern, falls der Kinderknoten zu leer geworden ist.

Der Baum kümmert sich darum, falls die Wurzel zu leer ist und löscht diese dann (nicht in ABTreeNode implementiert).

Aufgabe 11.4

```java
public class ABTreeNode {
  private int a;
  private int b;
  private int[] keys;
  private ABTreeNode[] children; // Only stores nodes of tree, list of all values is ommited

  // Constructor, insert, find, ...

  private void steal(ABTreeNode tooEmptyChild, ABTreeNode childToStealFrom) {
    // Already implemented
  }

  private void merge(ABTreeNode tooEmptyChild, ABTreeNode childToMergeWith, int keyFromParent) {
    // Already implemented
  }

  private int[] removeValueFromArray(int[] array, int index) {
    // Already implemented
  }

  public void delete(int key) {
    boolean isLowestRank = ____________________________;
    if (isLowestRank) {
```

```java



      return;
    }

    int childIndexToDeleteFrom = ____________________________;
    for (int i = 0; i < ____________________________; i++) {
      if (__________________________) {
        childIndexToDeleteFrom = i;
        break;
      }
    }





    // Prefer to merge with left and steal from right before checking other side
```

Aufgabe 11.4 | Teil 1: *delete* im Blatt

```java
public void delete(int key) {
  boolean isLowestRank = children.length == 0; // children == null also fine
  if (isLowestRank) {
    int indexToDelete = -1;
    for (int i = 0; i < keys.length; i++) {
      if (keys[i] == key) {
        indexToDelete = i;
      }
    }
    if (indexToDelete == -1) {
      return;
    }
    keys = removeValueFromArray(keys, indexToDelete);
    return;
  }
```

Aufgabe 11.4 | Teil 2: *delete* in innerer Node

```java

    int childIndexToDeleteFrom = keys.length;
    for (int i = 0; i < keys.length; i++) {
      if (keys[i] > key) {
        childIndexToDeleteFrom = i;
        break;
      }
    }

    ABTreeNode childToDeleteFrom = children[childIndexToDeleteFrom];
    childToDeleteFrom.delete(key);
```

Aufgabe 11.4 | Teil 3: *steal* und *merge*

```

    // Prefer to merge with left and steal from right before checking other side

    if (childToDeleteFrom.keys.length - 1 >= a) {
      return;
    }

    if (childIndexToDeleteFrom != children.length - 1 && children[childIndexToDeleteFrom + 1].keys.length >= a) {
      steal(childToDeleteFrom, children[childIndexToDeleteFrom + 1);
    } else if (childIndexToDeleteFrom != 0 && children[childIndexToDeleteFrom - 1].keys.length >= a) {
      steal(childToDeleteFrom, children[childIndexToDeleteFrom - 1);
    } else if (childIndexToDeleteFrom != 0) {
      merge(childToDeleteFrom, children[childIndexToDeleteFrom - 1, keys[childIndexToDeleteFrom - 1]);
    } else {
      merge(childToDeleteFrom, children[childIndexToDeleteFrom + 1, keys[childIndexToDeleteFrom]);
    }
  }
}
```