Datenstruktur Baum und Rekursion

Software Entwicklung 1

Annette Bieniusa, Mathias Weber, Peter Zeller

1 Datenstruktur Baum

> Bäume gehören zu den wichtigsten in der Informatik auftretenden Datenstrukturen.
> [Ottmann, Widmayer: Algorithmen und Datenstrukturen, 5. Auflage, Springer 2012]

Baumstrukturen finden sich in verschiedenen Kontexten in der Informatik wieder:

- Syntaxbäume, Darstellung von Termen, Dateisysteme, Darstellung von Hierarchien (z.B. Typhierarchie), Implementierung von Datenbanken etc.

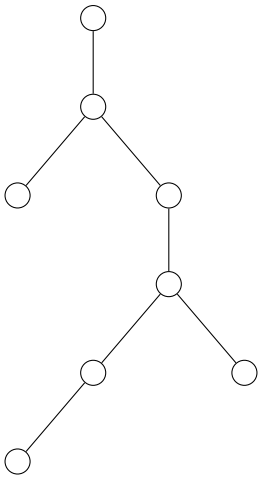
Führen wir zunächst einige Begriffe ein:

- In einem endlich verzweigten ***Baum*** hat jeder ***Knoten*** endlich viele ***Kinder***.
- Einen Knoten ohne Kinder nennt man ein ***Blatt***, einen Knoten mit Kindern einen ***inneren*** Knoten oder ***Zweig***.
- Den Knoten ohne Elternknoten nennt man ***Wurzel***.
- Zu jedem Knoten k gehört ein ***Unterbaum***, nämlich der Baum, der k als Wurzel hat.
- In einem ***Binärbaum*** hat jeder Knoten maximal zwei Kinder.

In unserer Vorlesung behandeln wir eine bestimmte Art von Bäumen, sogenannte *gerichtete gewurzelte* Bäume, bei denen wir stets einen Knoten als Wurzelknoten auszeichnen und die Verbindung von Eltern- zu Kindknoten (d.h. weg von der Wurzel) ausgerichtet ist. In der Graphentheorie werden allgemeinere Arten von Bäumen definiert, dazu erfahren Sie mehr in den Vorlesungen FGdP oder Graphentheorie.

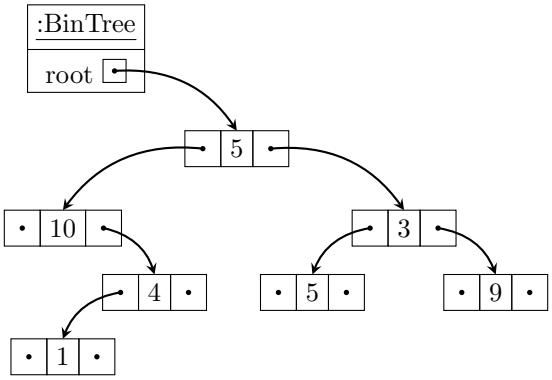
Frage 1:

- Markieren Sie im folgenden Baum einen Wurzelknoten!
- Welche Knoten sind Blätter? Welches sind innere Knoten?
- Handelt es sich um einen Binärbaum?



1.1 Markierte Bäume

Ein Baum heißt ***markiert***, wenn jedem Knoten k ein Wert/eine Markierung $m(k)$ zugeordnet ist.



Um einen markierten Binärbaum mit ganzen Zahlen als Markierung zu implementieren, benötigen wir zunächst eine Klasse für die Knoten. Diese erhält Attribute für die Markierung und die beiden Kindknoten `left` und `right`.

```
// Repraesentiert die Knoten eines Baums mit Markierungen
class TreeNode {
  private int mark;
  private TreeNode left;
  private TreeNode right;

  TreeNode(int mark, TreeNode left, TreeNode right) {
    this.mark = mark;
    this.left = left;
    this.right = right;
  }
  // Liefert die Markierung eines Knotens
  int getMark() {
    return this.mark;
  }
  // Liefert die Referenz auf den linken Kindknoten
  TreeNode getLeft() {
    return this.left;
  }
  // Liefert die Referenz auf den rechten Kindknoten
  TreeNode getRight() {
    return this.right;
  }
  ...
}

// Repraesentiert einen markierten Binaerbaum
public class BinTree {
  private TreeNode root;
  ...
}
```

Die Implementierung des Baums hält dann die Referenz auf den Wurzelknoten `root`. Von diesem Wurzelknoten sind alle anderen Knoten des Baumes erreichbar. Jeder Knoten des Baumes ist selbst wiederum Wurzelknoten des von ihm aufgespannten Unterbaums.
Bäume sind - wie auch Listen - ***rekursive Datentypen***. Bei rekursiven Datentypen wird der Datentyp selbst zu seiner eigenen Definition herangezogen.

- Eine Liste ist entweder leer oder besteht aus einem Element und einer (Rest-) Liste.
- Eine Baum ist entweder leer oder besteht aus einer Markierung und einem linken und rechten (Unter-) Baum.

Bei einer Definition einer rekursiven Datenstruktur gibt es einen oder mehrere Basisfälle (z.B. leere Liste / leerer Baum) und Rekursionsfälle, die beschreiben, wie man aus kleineren Instanzen der Datenstruktur größere aufbaut.
In der Implementierung in Java, die wir hier präsentieren, zeigt sich der rekursive Charakter von Bäumen dadurch, dass die Baumknoten als Attribute Referenzen auf Baumknoten haben.[1]

[1] In Java können Referenzen auf beliebige Objekt des zugehörigen Referenztyps verweisen. Dies kann

Berechnungen auf rekursiven Datenstrukturen wie Bäumen und Listen lassen sich in der Regel auf eine Kombination der Berechnung für den aktuellen Knoten und des Ergebnisses der Berechnung für den rekursiven Teil der Datenstruktur zurückführen. Essentiell ist es dabei, die Berechnung für die Basisfälle nicht zu vergessen.
Um z.B. alle Markierungen in einem mit int-markierten Binärbaum aufzusummieren, addieren wir zu der Markierung des Knotens, der diesen Baum aufspannt, die Summe der Markierungen für den linken und für den rechten Unterbaum. Dazu berechnen wir die Summe zum Unterbaum des linken Kindknoten, dann zum rechten Kindknoten und addieren die jeweiligen Ergebnisse dann zur Markierung des aktuellen Knotens. Im Basisfall für den leeren Baum (Knoten ist gleich null) ist die Summe gleich 0.

```
public class BinTree {
  private TreeNode root;
  ....
  // berechnet die Summe der Markierungen des Baums
  public int sum() {
    return sum(this.root);
  }
  private int sum(TreeNode node) {
    if (node == null) {
        return 0;
    }
    return node.getMark() + sum(node.getLeft())
                          + sum(node.getRight());
  }
```

Dabei verwenden wir die gleiche Methode `sum()` für die Summation bei den Kindknoten. Die Methode `sum()` ist rekursiv, sie ruft sich selbst erneut auf. Die rekursive Struktur des Baums spiegelt sich wieder in diesen rekursiven Methodenaufrufen.

Frage 2: Wie können wir testen, ob der Baum eine bestimmte Markierung enthält?

- Was ist das Ergebnis im Basisfall (leerer Baum)?
- Was ist das Ergebnis im Rekursionsfall?

Schreiben Sie eine Methode `public boolean contains(int x)` für die Klasse `BinTree`, die `true` liefert, wenn einer der Knoten des Baumes mit `x` markiert ist!

Mehr zur Rekursion erfahren Sie in Abschnitt 2.

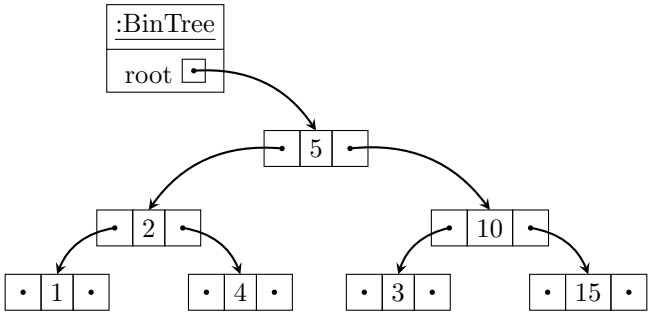
zu unerwünschten zirkulären Objektgeflechten führen, die der obigen Definition von rekursiven Datentypen widersprechen. Wir gehen hier davon aus, dass innerhalb der hierbeschriebenen rekursiven Datentypen unterschiedliche Referenzen nicht auf das gleiche Objekt verweisen.

1.2 Sortierte markierte Bäume

Definition Ein mit ganzen Zahlen markierter Binärbaum heißt ***sortiert***, wenn für alle Knoten k gilt:

- Alle Markierungen der linken Nachkommen von k sind kleiner als oder gleich $m(k)$.
- Alle Markierungen der rechten Nachkommen von k sind größer als $m(k)$.

Frage 3: Ist dieser markierte Binärbaum sortiert?



Die Sortiertheitseigenschaft von sortierten markierten Binärbaum erleichtert es ungemein, Elemente in dem Baum zu suchen. Sie werden daher auch oft ***Suchbäume*** genannt.
Wir beginnen wieder mit dem `root`-Knoten:

- In einem leeren Baum ist das Element nicht enthalten.
- Falls der aktuelle Knoten mit dem gesuchten Element markiert ist, ist es im Baum enthalten.
- Andernfalls suchen wir rekursiv beim linken Kindknoten, falls das gesuchte Element kleiner ist, als die Markierung des Knotens; ist das gesuchte Element größer, suchen wir rekursiv beim rechten Kindknoten.

```
// Repraesentiert einen sortierten markierten Binaerbaum
public class SortedBinTree {
  private TreeNode root;
  public SortedBinTree() {
    root = null;
  }
  // prueft, ob ein Element im Baum enthalten ist
  public boolean contains(int element) {
    return contains(root, element);
  }
  private boolean contains(TreeNode node, int element) {
    if (node == null) {
```

```
            return false;
        }
        if (element < node.getMark()) {
            // kleinere Elemente links suchen
            return contains(node.getLeft(), element);
        } else if (element > node.getMark()) {
            // groessere Elemente rechts suchen
            return contains(node.getRight(), element);
        } else {
            // gefunden!
            return true;
        }
    }
```

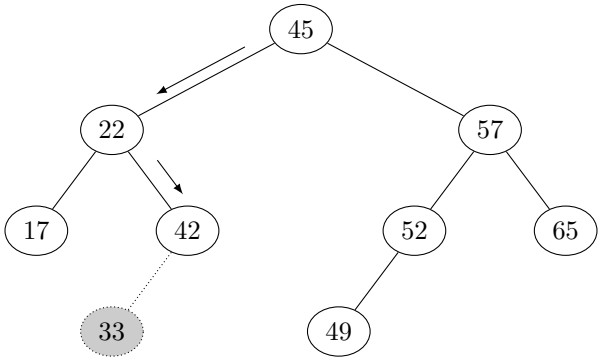
Neue Knoten werden immer als Blätter eingefügt. Die Position des Blattes wird durch den Wert des neuen Eintrags festgelegt. Um Elemente in einen `SortedBinTree` einzufügen wählen wir das folgende algorithmische Vorgehen:

- Ist der Baum noch leer, wird der erste Eintrag als Wurzel des Baums hinzugfügt.
- Ein Knoten wird
  - in den linken Unterbaum der Wurzel eingefügt, wenn sein Wert kleiner gleich ist als der Wert der Wurzel;
  - in den rechten, wenn er größer ist.

  Dieses Verfahren wird rekursiv fortgesetzt, bis die Einfügeposition bestimmt ist.

Beispiel: Beim Einfügen von 33 in den skizzierten Baum wird - angefangen bei der Wurzel - die Markierung des Knotens mit 33 verglichen.

- 33 ist kleiner als 45; daher wird das Einfügen rekursiv auf dem linken Kindknoten fortgeführt;
- 33 ist größer als 22; daher wird das Einfügen rekursiv auf dem rechten Kindknoten fortgeführt;
- 33 ist kleiner als 42; da der Knoten mit Markierung 42 keinen linken Kindknoten hat, wird ein neuer Knoten mit Markierung 33 als linker Kindknoten hier eingefügt.



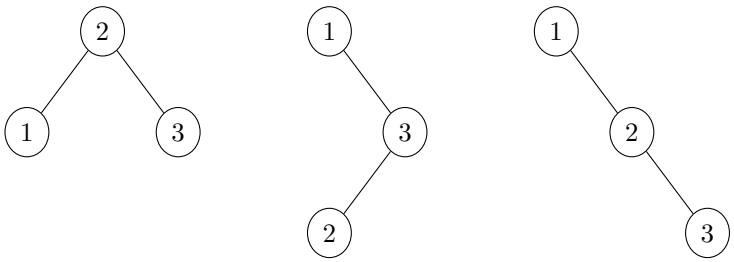
Die Implementierung folgt diesem rekursiven Schema:

```
public void add(int element) {
    root = add(root, element);
}

private TreeNode add(TreeNode node, int element){
    if (node == null) {
        return new TreeNode(element);
    } else if (element <= node.getMark()) {
        node.setLeft(add(node.getLeft(), element));
    } else {
        node.setRight(add(node.getRight(), element));
    }
    return node;
}
```

Bemerkungen

- Die Reihenfolge des Einfügens bestimmt das Aussehen des sortierten markierten Binärbaums.

  Die folgenden Bäume sind durch die Einfügereihenfolge $2 \to 3 \to 1$ bzw. $1 \to 3 \to 2$ bzw. $1 \to 2 \to 3$ enstanden.



- Wie man bei dem letzten der drei Beispiel sieht, entartet der Baum bei sortierter Einfügereihenfolge zur linearen Liste.

2 Rekursion

Rekursion ist eine elegante Strategie zur Problemlösung, die es erlaubt eine Problemstellung auf ein ähnliche, aber kleinere Problemstellung zurückzuführen. Um Rekursion genauer zu charakterisieren, führen wir hier einige Begriffe ein. Diese beziehen sich auf Methoden, sind aber analog auf Datenstrukturen, Prozeduren etc. anwendbar.

Definition Eine Methodendeklaration m heißt ***direkt rekursiv***, wenn der Methodenrumpf einen Aufruf von m enthält.
Eine Menge von Methodendeklarationen heißen ***verschränkt rekursiv*** oder ***indirekt rekursiv*** (engl. *mutually recursive*), wenn die Deklarationen gegenseitig voneinander abhängen.
Eine Methodendeklaration heißt ***rekursiv***, wenn sie direkt rekursiv ist oder Element einer Menge verschränkt rekursiver Methoden ist.

Wie bereits im vorherigen Abschnitt gesehen, tritt Rekursion in natürlicher Weise auf Datenstrukturen wie Listen und Bäumen auf. Wir können auch aber auch rekursive Methoden verwenden, um Berechnungen auf Zahlen durchzuführen. Das klassiche Beispiel für eine rekursive Prozedur ist die Berechnung der Fakultätsfunktion, die das Produkt der ersten n natürlichen Zahlen berechnet.

$$n! = \begin{cases} n * (n-1)! & \text{falls } n > 0 \\ 1 & \text{falls } n = 0 \end{cases}$$

```
public static int fac(int n) {
   if (n == 0) {
      return 1;
   } else {
      return n * fac (n-1);
   }
}
```

- Der ***Basisfall*** liefert einen Wert ohne einen nachfolgenden rekursiven Aufruf.
- Der ***Rekursionsschritt*** verwendet das Ergebnis von rekursiven Methodenaufrufen für (andere) Parameterwerte für die Berechnung im aktuellen Methodenaufruf.
- *Wichtig:* Der Basisfall muss nach endlich vielen rekursiven Aufrufen erreicht werden, damit das Programm terminiert.
  Dies ist in dieser Implementierung nur für Parameter $n \geq 0$ der Fall[2].

Eine rekursive Methodendeklaration m heißt ***linear rekursiv***, wenn in jedem Ausführungszweig höchstens ein Aufruf von m auftritt. Die Methode zur Implementierung der Fakultätsfunktion ist ein typisches Beispiel für eine linear rekursive Methode.

Kaskadenartige Rekursion Die Fibonacci-Folge ist eine Folge von natürlichen Zahlen, die bei diversen Naturphänomen in Erscheinung tritt (z.B. zur Modellierung von Hasenpopulationen). Jede Fibonacci-Zahl ist die Summe ihrer beiden Vorgänger:

$$0 \quad 1 \quad 1 \quad 2 \quad 3 \quad 5 \quad 8 \quad 13 \quad 21 \quad 34 \quad 55 \quad 89 \quad 144 \quad 233 \ldots$$

$$fib(n) = \begin{cases} 0 & \text{für } n = 0 \\ 1 & \text{für } n = 1 \\ fib(n-1) + fib(n-2) & \text{für } n \geq 2 \end{cases}$$

```
public static int fib(int n) {
  if (n == 0) {
    return 0;
  }
  if (n == 1) {
```

[2]Wir lassen in diesem Fall Überläufe außer Acht.

```
    return 1;
  }
  return fib(n-2) + fib(n-1);
}
```

Eine rekursive Methodendeklaration m, die nicht linear rekursiv ist, heißt ***kaskadenartig rekursiv***. Das heißt, es muss in mindestens einem Ausführungszweig der Methode mehr als ein Aufruf von m auftreten.

Repetitive Rekursion Eine linear rekursive Methodendeklaration m heißt ***repetitiv rekursiv*** (auch endrekursiv, engl. *tail recursive*), wenn jeder Aufruf von m in m der letzte auszuwertende Ausdruck ist.
Die Prozedur `boolean contains(TreeNode node, int element)` aus Abschnitt 1.2 ist repetitiv-rekursiv, da der Aufruf von `contains()` der jeweils letzte auszuwertende (Teil-) Ausdruck ist. Die Prozedur `fib()` ist hingegen nicht repetitiv rekursiv: Hier ist der letzte auszuwertende Ausdruck die Addition der Rückgabewerte der rekursiven Aufrufe.

> **Frage 4:** Auch die Prozedur zur Berechnung der Fakultätsfunktion ist in der obigen Implementierung nicht repetitiv-rekursiv. Wie kann man sie umschreiben, um eine repetitiv-rekursive Variante zu erhalten.

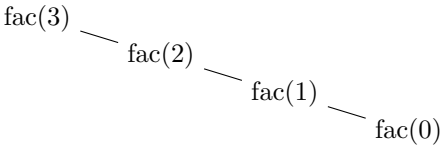
Die Ausführung von repetitiv-rekursive Methoden ist in einigen Programmiersprachen sehr effizient umgesetzt (z.B. Haskell) und dort von besonderer Bedeutung.

Verschränkte Rekursion Zuletzt nochmal ein Beispiel zur verschränkten Rekursion. Das folgende Methodenpaar testet, ob eine (positive) Zahl gerade bzw. ungerade ist.

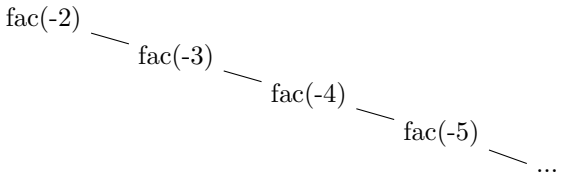
```
public static boolean istGerade(int n) {
  if (n == 0) {
    return true;
  } else {
    return istUngerade(n-1);
  }
}

public static boolean istUngerade(int n){
  if (n == 0) {
    return false;
  } else {
    return istGerade(n-1);
  }
}
```

2.1 Zur Terminierung von rekursiven Funktionen

Angenommen, wir wollen die Fakultät von 3 berechnen. Dabei wird folgender Prozeduraufrufbaum erzeugt:



Was geschieht aber beim Aufruf von `fac(-2)`? Diese Ausführung liefert folgenden Prozeduraufrufbaum:



Die Ausführung für den Parameterwert -2 liefert eine unendliche Folge von rekursiven Aufrufen von `fac()`: Die Ausführung terminiert nicht[3], da der Basisfall mit Parameterwert 0 nicht erreicht wird. Es lässt sich dabei folgende Beobachtung machen: Der Abstand der Parameterwerte zum Basisfall wird beim Aufruf mit Parameterwert 3 mit jedem Aufruf kleiner, beim Aufruf mit Parameterwert -2 jedoch mit jedem Aufruf größer.
Die Ideen den Abstand zum Basisfall zu "messen" kann man verwenden, um zu beweisen, dass eine Methode terminiert. Dazu definiert man eine sogenannte Abstiegsfunktion h, welche den Abstand der Parameterwerte zu einem Basisfall misst. Dazu muss h jeder gültigen Kombination von Parameterwerten eine natürliche Zahl ($\mathbb{N}_0$) zuweisen. Es sollte dann gelten, dass der Wert von h bei jedem rekursiven Aufruf für die neuen Parameterwerte echt kleiner ist als für die Parameterwerte des aktuellen Aufrufs. Da in $\mathbb{N}_0$ jede monoton fallende Folge endlich ist, ist eine unendliche Folge von rekursiven Aufrufen durch die Voraussetzungen ausgeschlossen. Wenn andere Ursachen zur Nichtterminierung (zum Beispiel Schleifen) ausgeschlossen werden können, dann ist damit die Terminierung der rekursiven Methode gezeigt.

```
int f(int x, int y) {
  if (x >= y) {
    return x;
  } else {
    return f(2*x, y);
  }
}
```

Die Methode `f(x, y)` berechnet die kleinste Zahl $z \geq y$, so dass $z = x \cdot 2^a$ für ein $a \in \mathbb{N}_0$. Wir wollen zeigen, dass die Methode für Werte $x > 0$ terminiert. Die Idee ist, dass dann x in jedem rekursiven Aufruf größer wird und somit näher an den Basisfall $x >= y$ kommt. Dazu definieren wir die Abstiegsfunktion $h(x, y) = max(0, y - x)$. Wir

[3] In Java benötigen Methodenaufrufe Speicherplatz, so dass in der Praxis nach endlich vielen Aufrufen der Speicher voll ist und der Fehler `StackOverflowError` ausgelöst wird.

verwenden hier das Maximum mit 0 um auszudrücken, dass der Basisfall für $x \geq y$ bereits erreicht ist.
Für den rekursiven Aufruf müssen wir jetzt zeigen, dass $h(2 \cdot x, y) < h(x, y)$ gilt, also dass die neuen Parameterwerte einen kleineren Wert der Abstiegsfunktion haben. Dies ist hier der Fall, da $2 \cdot x$ mit den gewählten Voraussetzungen immer größer als x ist[4].
Dieses Nachweisverfahren zur Terminierung ist nicht auf Methoden mit Zahlen beschränkt. Um die Terminierung von rekursiven Methoden auf Listen beispielsweise nachzuweisen, kann man die Länge der Liste zur Konstruktion von h zur Hilfe nehmen.

2.2 Rekursion vs. Iteration

Für jede Methode, die ein Problem mit Hilfe von Rekursion löst, kann eine Alternativimplementierung gefunden werden, die das gleiche Problem iterativ, d.h. mit Schleifenkonstrukten löst.

```
public static int fac(int n) {
   if (n == 0) {
      return 1;
   } else {
      return n * fac (n-1);
   }
}
```

```
public static int fac(int n) {
   int result = 1;
   int i = n;
   while (i != 0) {
    result = i * result;
    i--;
   }
   return result;
}
```

```
class IntList {
  private Node first;
  ...
  public int sumEntries() {
    return sumEntries(this.first);
  }
  private int sumEntries(Node n) {
    if (n == null) {
      return 0;
    } else {
      return n.getValue()
           + sumEntries(n.getNext());
    }
  }
}
```

```
class IntList {
  private Node first;
    ...
  public int sumEntries() {
    int result = 0;
    Node n = this.first;
    while (n != null) {
      result += n.getValue();
      n = n.getNext();
    }
    return result;
  }
```

Die rekursive Variante führt allerdings oft zu einfacherem und eleganterem Code, wenn man Probleme auf rekursiven Datenstrukturen löst.
Manche Programmiersprachen (insbesondere funktionale Sprachen wie Haskell, Ocaml, Erlang etc.) bieten keine Sprachmittel für Schleifen an. In diesen Sprachen müssen rekursive Funktionen immer dann verwendet werden, wenn die Anzahl der Operationen

[4] Streng genommen sollten wir noch y einschränken, zum Beispiel $y < 2^{30}$, um Überläufe ausschließen zu können.

nicht von vorneherein beschränkt ist[5]. Andererseits gibt es einige imperative Sprachen (OpenCL, diverse C Compiler für Mikrocontroller, ältere Versionen von Fortran), die wiederum keine Rekursion als Sprachmittel zur Verfügung stellen. Programmierer müssen daher beide Varianten verstehen und anwenden können.

[5] Oft verwendet man in diesen Sprachen auch Funktionen höherer Ordnung wie `map`, `filter` und `fold`, welche von bestimmten rekursiven Anwendungsfällen abstrahieren.

Hinweise zu den Fragen

Hinweise zu Frage 1: Jeder Knoten kann hier als Wurzelknoten betrachtet werden. Dementsprechend ergibt sich die Antwort auf die anderen beiden Teilfragen.

Hinweise zu Frage 2: Im Basisfall (leerer Baum → `n == null`) kann `x` nicht enthalten sein. Daher wird hier `false` zurückgegeben. Andernfalls testen wir, ob der aktuelle Knoten selbst die Markierung hat oder ob sich die Markierung im Unterbaum von seinem linken bzw. rechten Kindknoten findet.

```
class BinTree {
    ...
    public boolean contains(int x) {
      return contains(x, this.root);
    }
    private boolean contains(int x, TreeNode n){
      if (n == null) {
        return false;
      } else {
        return n.getMark() == x
          || contains(x, n.getLeft())
          || contains(x, n.getRight());
      }
    }
}
```

Hinweise zu Frage 3: Nein, rechts vom Knoten mit Markierung 5 gibt es einen Knoten mit Markierung 3.

Hinweise zu Frage 4:

```
public static int fac(int n) {
   return helperFac(n, 1);
}
public static int helperFac(int n, int x) {
   if (n == 0) {
      return x;
   } else {
      return helperFac(n-1, n * x);
   }
}
```