Tutoraufgabe 1 (Implementierung eines ADTs):

Wir spezifizieren den ADT Union-Find wie folgt:

Wertebereich: Sei `Element` ein Datentyp, der eine ganze Zahl in seinem Feld `key` enthält, d. h. `e.key` ist vom Typ `int` für `e` vom Typ `Element`.

Der Wertebereich des ADT Union-Find enthält für alle $n \geq 1$ alle Partitionen von $\{\mathtt{e}_1, \ldots, \mathtt{e}_n\}$ mit $\mathtt{e}_i\mathtt{.key} = i$ für alle $i \in \{1, \ldots, n\}$. D. h. der Wertebereich enthält für alle $n \geq 1$ und $1 \leq k \leq n$ alle Mengen $\{T_1, \ldots, T_k\}$ mit $T_i \subseteq \{\mathtt{e}_1, \ldots, \mathtt{e}_n\}$, $\mathtt{e}_i\mathtt{.key} = i$ für alle $1 \leq i \leq n$, $T_i \neq \emptyset$ für alle $1 \leq i \leq k$, $T_i \cap T_j = \emptyset$ für alle $1 \leq i < j \leq k$ und

$$\bigcup_{i=1}^{k} T_i \;=\; \{\mathtt{e}_1, \ldots,\ \mathtt{e}_n\}\,.$$

Eine Menge in einer Partition wird durch eins ihrer Elemente, ihren eindeutigen Repräsentanten, identifiziert.

Operationen:

- Der Konstruktor `UnionFind(int n)` erstellt die Partition $\{\{\mathtt{e}_1\}, \ldots, \{\mathtt{e}_n\}\}$ von $\{\mathtt{e}_1, \ldots, \mathtt{e}_n\}$ mit $\mathtt{e}_i\mathtt{.key} = i$ für alle $1 \leq i \leq n$. Vorbedingung: $\mathtt{n} \geq 1$.
- `Element find(Element e)` gibt den Repräsentanten derjenigen Menge T_j aus der Partition zurück, die `e` enthält. Vorbedingung: $1 \leq \mathtt{e.key} \leq n$.
- `void union(Element e1, Element e2)` vereinigt in der Partition die Mengen mit den Repräsentanten `e1` und `e2`. Vorbedingung: Die Partition enthält Mengen mit Repräsentanten `e1` und `e2`.

(Um diese Datenstruktur sinnvoll verwenden zu können, benötigen wir noch eine Methode, die für ein $1 \leq i \leq n$ das in der Datenstruktur verwendete Element `e` mit $\mathtt{e.key} = i$ zurückgibt; um die Aufgabe nicht zu komplex werden zu lassen, vernachlässigen wir diese Methode in der Schnittstelle.)

Beispiel: Wir initialisieren eine Union-Find Datenstruktur `UnionFind uf = UnionFind(5)` und erhalten

e1	e2	e3	e4	e5
1	2	3	4	5

Führen wir nun `uf.union(e1, e3)` aus, erhalten wir

e1	e3	e2	e4	e5
1	3	2	4	5

wobei `uf.find(e1) == uf.find(e3) == e1`. Führen wir zudem `uf.union(e5, e3)` aus, erhalten wir

e1	e3	e5	e2	e4
1	3	5	2	4

wobei `uf.find(e5) == e1`.

a) Implementieren Sie die Union-Find Datenstruktur unter Nutzung *genau einer Instanz* einer wie in der Vorlesung vorgestellten **doppelt verketteten Liste** (`List`). Es dürfen jedoch beliebig viele primitive (elementare) Datentypen verwendet werden. *Tipp: Benutzen Sie zum Trennen der Mengen in einer Partition ein Element mit dem Wert 0.*

b) Implementieren Sie die Union-Find Datenstruktur unter Nutzung *genau einer Instanz* `t` eines (nicht reduzierten) **doppelt verketteten Baumes** (`Tree`), welcher wie der in der Vorlesung vorgestellte Baum definiert ist, nur dass jedes `Element` (Knoten) `e` zusätzlich einen Vorgänger `e.father` hat (mit `t.root.father == null` und `e.father != null` für `e != t.root`).

Sie können die folgenden Operationen für `Tree` verwenden:

- `void addChild(Element e, int key)` fügt dem gegebenen Element (Knoten) `e` ein Kind mit Wert `key` hinzu. Vorbedingung: `e` ist ein Knoten im Baum.
- `void removeChild(Element e1, Element e2)` entfernt das Kind `e2` von `e1`. Vorbedingung: `e1` ist ein Knoten im Baum, `e2` ist ein Kind von `e1`.
- `void appendChild(Element e1, Element e2)` schneidet `e2` von seinem Vater ab und fügt es anschließend `e1` als Kind hinzu. Vorbedingung: `e1` und `e2` befinden sich im Baum und `e2` ist kein Vorgänger von `e1` (befindet sich nicht auf dem Pfad von der Wurzel zu `e1`).

Es dürfen jedoch beliebig viele primitive (elementare) Datentypen verwendet werden. *Tipp: Benutzen Sie zum Trennen der Mengen in einer Partition ein Element mit dem Wert 0.*

Lösung:

a) Die Liste fängt mit einem Element mit Schlüssel 0 an. Danach stehen die Elemente der Teilmengen, getrennt durch jeweils ein Element mit Schlüssel 0. Die Teilmengen werden repräsentiert durch das erste Element in dem entsprechenden Block in der Liste.

```
class UnionFind
{
// Attribute
List content;

// Operationen
UnionFind(int n)
{
    content = List(); // konstruiert eine leere Liste
    content.insert(0);
    for(int i = 1; i <= n; i++)
    {
        content.insert(i);
        content.insert(0);
    }
}

Element find(Element e)
{
    Element representative = e;
    while(content.predecessor(representative).key != 0)
    {
        representative = content.predecessor(representative);
    }
    return representative;
}

void union(Element e1, Element e2)
{
    if(e1 == e2) return; // die Teilmengen sind identisch
    // finde den Separator nach der 1. Teilmenge
    Element post1 = content.successor(e1);
    while(post1.key != 0)
    {
        post1 = content.successor(post1);
    }
    //finde den Separator vor der 2. Teilmenge
    Element pre2 = content.predecessor(e2);
    // finde den Separator nach der 2. Teilmenge
    Element post2 = content.successor(e2);
    while(post2.key != 0)
```

```
    {
        post2 = content.successor(post2);
    }
    // nehme die 2. Teilmenge (e2 bis post2) aus der Liste
    pre2.next = post2.next;
    if (post2.next != null) post2.next.prev = pre2;
    else tail = pre2;
    // fuege die 2. Teilmenge (e2 bis post2) hinter der 1. Teilmenge (post1) ein
    post2.next = post1.next;
    if(post1.next != null) post1.next.prev = post2;
    else tail = post2;
    post1.next = e2;
    e2.prev = post1;
    // entferne den Separator zwischen den beiden Teilmengen
    content.remove(post1);
}
}
```

b) Der Baum hat eine Wurzel mit Schlüssel 0. Die Teilmengen sind die Kinder der Wurzel.

```
class UnionFind
{
// Attribute
Tree content;

// Operationen
UnionFind(int n)
{
    // konstruiere einen Baum mit einem Knoten (=root) mit dem Wert (key) 0
    content = Tree(0);
    // fuege fuer jedes 1<=i<=n ein Kind an
    for(int i = 1; i <= n; i++)
    {
        content.addChild(content.root, i);
    }
}

Element find(Element e)
{
    Element representative = e;
    while(representative.father.key != 0)
    {
        representative = representative.father;
    }
    return representative;
}

void union(Element e1, Element e2)
{
    if(e1 == e2)
    {
        return; // die Teilmengen sind identisch
    }
    // entferne den 2. Repraesentanten von der Wurzel
    // und fuege ihn dem 1. Repraesentanten als Kind hinzu
    appendChild(e1, e2);
}
}
```

Aufgabe 2 (Implementierung eines ADTs): (3+3=6 Punkte)

Betrachten Sie einen Ringspeicher einer festen Größe k, welcher durch folgende Spezifikation definiert ist:

Wertebereich: Speicher der Kapazität k, d. h. er kann maximal k Elemente mit einem Schlüssel (im Folgenden Wert genannt) vom Typ `int` enthalten.

Operationen:

- Der Konstruktor `CircularBuffer(int k)` erstellt einen leeren Ringspeicher der Kapazität k. Vorbedingung: $k > 1$.
- `bool isEmpty()` prüft ob der Ringspeicher leer ist.
- `bool isFull()` prüft ob der Ringspeicher voll ist.
- `int read()` gibt den Wert des ältesten Elementes aus dem Ringspeicher zurück und löscht das Element anschließend. Die anderen Elemente werden nicht verändert. Vorbedingung: der Ringspeicher ist nicht leer.
- `void write(int key)` schreibt ein Element mit dem gegebenen Wert in den Ringspeicher. Ist der Ringspeicher voll, so wird das älteste Element überschrieben. Die vorigen Elemente im Speicher (bis auf das älteste falls der Speicher voll ist) bleiben unverändert.

Beispiel: Wir initialisieren mit `CircularBuffer cb = CircularBuffer(5)` einen leerer Ringspeicher der Kapazität 5. Führt man `cb.write(2)`, `cb.write(6)`, `cb.write(4)` und `cb.write(3)` in dieser Reihenfolge aus, erhält man die Belegung (2, 6, 4, 3), wobei das linke Element mit dem Wert 2 das älteste ist. Dann ist `cb.read() = 2` und führt zum Speicher (6, 4, 3). Führt man des Weiteren `cb.write(7)` und `cb.write(1)` aus, so erhält man den vollen Speicher (6, 4, 3, 7, 1). Ein weiteres Ausführen von `cb.write(8)` überschreibt die Speicherzelle mit `6` und führt also zu (4, 3, 7, 1, 8).

a) Implementieren Sie den Ringspeicher unter Nutzung genau einer Instanz des in der Vorlesung vorgestellten **Arrays** `int[]`. Es dürfen jedoch beliebig viele primitive (elementare) Datentypen verwendet werden.

b) Implementieren Sie den Ringspeicher unter Nutzung genau einer Instanz der in der Vorlesung vorgestellten **einfach verketteten Liste** (`List`) mit den folgenden Operationen:

- `Element insert(Key k)` fügt ein Element mit Schlüssel `k` am Anfang der Liste ein.
- `void remove(Element x)` entfernt das Element `x` aus der Liste. Vorbedingung: `x` ist in der Liste.
- `Element successor(Element x)` gibt das Nachfolgerelement von Element `x` in der Liste zurück. Vorbedingung: `x` ist in der Liste.
- `Element predecessor(Element x)` gibt das Vorgängerelement von Element `x` in der Liste zurück. Vorbedingung: x ist in der Liste.
- `int length()` gibt die Länge der Liste zurück.

Es dürfen jedoch beliebig viele primitive Datentypen verwendet werden.

Lösung:

a)
```
class CircularBuffer
{
    // Attribute
    int[] content;
    int first; // der aelteste Wert
    int last; // die Speicherzelle in die geschrieben werden soll
    int capacity;

// Operationen
```

```
CircularBuffer(int k)
{
    content = int[k+1]; // reserviere k+1 Speicherzellen
    // wir lassen immer eine Speicherzelle frei um leere und volle
    // Ringspeicher unterscheiden zu koennen
    first = last = 0;
    capacity = k + 1;
}

bool isEmpty()
{
    return first == last;
}

bool isFull()
{
    return (last + 1 % capacity) == first; // % ist die Modulo-Operation
}

int read()
{
    int result = content[first];
    first = first + 1 % capacity;
    return result;
}

void write(int key)
{
    content[last] = key;
    last = last + 1 % capacity;
    if(first == last)
    {
        first = first + 1 % capacity;
    }
}
}
```

b) Folgende Lösung stellt eine alternative und in der Praxis eher unübliche Variante eines Ringspeichers da. Eine bessere Lösung mit einer Liste erhält man, wenn man analog zur Lösung von a) vorgeht.

```
class CircularBuffer
{
    // Attribute
    List content;
    Element first;
    int capacity;

// Operationen
CircularBuffer(int k)
{
    content = List();
    first = null;
    capacity = k;
}

bool isEmpty()
{
    return first == null;
}

bool isFull()
{
```

```
        return content.length() == capacity;
    }

    int read()
    {
        Element oldest = first;
        while (oldest.next != null)
        {
            oldest = oldest.next;
        }
        if (first == oldest) first = null;
        int result = oldest.key;
        content.remove(oldest);
        return result;
    }

    void write(int key)
    {
        if(isFull())
        {
            read();
        }
        Element newElement = content.insert(key);
        if (first == null) first = newElement;
    }
    }
```

Tutoraufgabe 3 (ADT einer Implementierung):

Gegeben sei die folgende Klasse in der Programmiersprache Java:

```
public class A {

    private int[] pos;
    private int[] neg;

    public A() {
        this.pos = new int[67108864];
        this.neg = new int[67108864];
    }

    public boolean contains(int i) {
        if (i < 0) {
            final int j = -i - 1;
            return (this.neg[j / 32] & (1 << (j % 32))) != 0;
        } else {
            return (this.pos[i / 32] & (1 << (i % 32))) != 0;
        }
    }

    public void add(int i) {
        if (i < 0) {
            final int j = -i - 1;
            this.neg[j / 32] |= (1 << (j % 32));
        } else {
            this.pos[i / 32] |= (1 << (i % 32));
        }
```

```
    }

    public void remove(int i) {
        if (i < 0) {
            final int j = -i - 1;
            this.neg[j / 32] &= (-(1 << (j % 32)) - 1);
        } else {
            this.pos[i / 32] &= (-(1 << (i % 32)) - 1);
        }
    }

}
```

Geben Sie eine möglichst allgemeine Spezifikation (Wertebereich und Signatur) für den abstrakten Datentyp an, der durch diese Klasse implementiert wird. Dabei ist der Wertebereich des Datentyps `int` die Menge $I = \{-2^{31}, -2^{31}+1, \ldots, 2^{31}-1\}$.

Lösung:

Ein Objekt vom Typ `A` repräsentiert eine Menge ganzer Zahlen aus I. Der Wertebereich des ADTs, den `A` implementiert, ist also 2^I. Dieser ADT bietet die folgenden Operationen an:

- Der Konstruktor `A()` erstellt die leere Menge.
- `boolean contains(int i)` gibt `true` zurück, falls die Zahl `i` in der Menge enthalten ist. Ansonsten gibt es `false` zurück.
- `void add(int i)` fügt die Zahl `i` zur Menge hinzu.
- `void remove(int i)` löscht die Zahl `i` aus der Menge.

Aufgabe 4 (ADT einer Implementierung): (4 Punkte)

Gegeben seien folgende zwei Klassen in der Programmiersprache Java:

```
public class B {

  private Node head;

  public B() {
    this.head = null;
  }

  public Object get(int i) {
    Node c = this.head;
    while (c != null) {
      int j = c.getKey();
      if (j == i) {
        return c.getValue();
      } else if (j < i) {
        c = c.getRight();
      } else {
        c = c.getLeft();
      }
    }
    return null;
  }

  public void put(int i, Object o) {
    if (this.head == null) {
      this.head = new Node(i, o);
    } else {
      Node c = this.head;
      while (true) {
        int j = c.getKey();
        if (j == i) {
          c.setValue(o);
          return;
        } else if (j < i) {
          if (c.getRight() == null) {
            c.setRight(new Node(i, o));
            return;
          }
          c = c.getRight();
        } else {
          if (c.getLeft() == null) {
            c.setLeft(new Node(i, o));
            return;
          }
          c = c.getLeft();
        }
      }
    }
  }

}
```

```
public class Node {

    private final int key;
    private Node left;
    private Node right;
    private Object value;

    public Node(int i, Object o) {
        this.left = null;
        this.right = null;
        this.key = i;
        this.value = o;
    }

    public int getKey() {
        return this.key;
    }

    public Node getLeft() {
        return this.left;
    }

    public Node getRight() {
        return this.right;
    }

    public Object getValue() {
        return this.value;
    }

    public void setLeft(Node c) {
        this.left = c;
    }

    public void setRight(Node c) {
        this.right = c;
    }

    public void setValue(Object o) {
        this.value = o;
    }

}
```

Geben Sie eine möglichst allgemeine Spezifikation (Wertebereich und Signatur) für den abstrakten Datentyp an, der durch die Klasse `B` implementiert wird. Die Menge aller Objekte (inklusive `null`) sei hierbei mit *Obj* bezeichnet (d. h. der Wertebereich des Datentyps `Object` ist *Obj*).

Lösung:

Ein Objekt vom Typ `B` repräsentiert eine Funktion von den ganzen Zahlen zwischen -2^{31} und $2^{31}-1$ zur Menge aller Objekte (inklusive `null`). Der Wertebereich des ADTs, den `B` implementiert, ist also die Menge $\{f : \texttt{int} \to Obj\}$ aller solchen Funktionen. Dieser ADT bietet die folgenden Operationen an:

- Der Konstruktor `B()` erstellt eine Funktion, die allen Zahlen `null` zuordnet.
- `Object get(int i)` gibt das Objekt zurück, das der Zahl `i` durch die Funktion zugeordnet wird.
- `void put(int i, Object o)` ordnet der Zahl `i` das Objekt `o` zu.

Tutoraufgabe 5 (ADT verwenden):

Gegeben sei die folgende Spezifikation des abstrakten Datentyps `BinaryTree` für binäre Bäume:

Wertebereich: Spezifiziert durch alle möglichen Belegungen der Variablen
`int value`, `BinaryTree leftChild` und `BinaryTree rightChild`

Operationen:

- `int value()` gibt den Wert von `value` zurück.
- `BinaryTree left()` gibt den Wert von `leftChild` zurück.
- `BinaryTree right()` gibt den Wert von `rightChild` zurück.

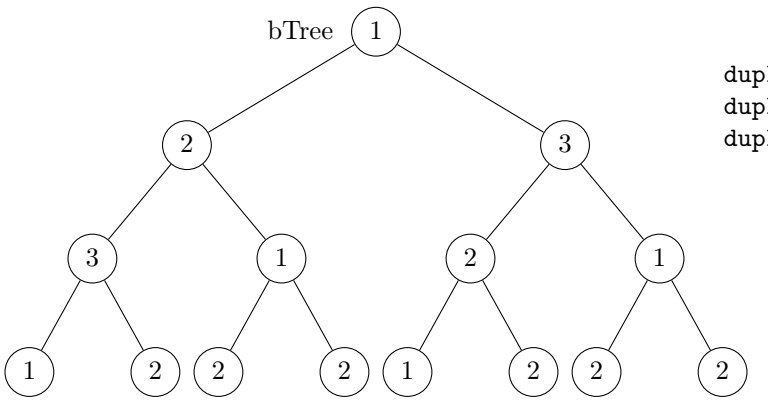
Ein Pfad in einem Binärbaum `b` $\neq$ `null` vom Typ `BinaryTree` ist eine Sequenz $\texttt{e}_1, \ldots, \texttt{e}_n$ von `BinaryTree` Objekten mit $n \geq 1$, $\texttt{e}_1 = \texttt{b}$, $\texttt{e}_{i+1} \in \{\texttt{left(e}_i\texttt{)}, \texttt{right(e}_i\texttt{)}\}$ für alle $1 \leq i \leq n-1$ und $\texttt{left(e}_n\texttt{)} = \texttt{right(e}_n\texttt{)} = \texttt{null}$.
Entwerfen Sie

a) einen rekursiven Algorithmus `bool duplicate(BinaryTree b, int v, bool found)`,

b) einen iterativen Algorithmus `bool duplicate(BinaryTree b, int v)`,

der überprüft, ob es im Binärbaum `b` einen Pfad gibt, auf dem mindestens zwei Objekte den Wert `v` haben. Verwenden Sie Stapel(`Stacks`) für den iterativen Algorithmus.

Beispiel:




```
duplicate(bTree,1) = true
duplicate(bTree,2) = true
duplicate(bTree,3) = false
```

Lösung:

a)
```
boolean duplicate(BinaryTree b, int v, boolean found)
{
  boolean equals = (b.value() == v);
  if (equals && found) return true; // first instance already found
  found = found || equals;
  if (b.left() != null)
    if (duplicate(b.left(),v,found))
      return true;
  if (b.right() != null)
    return duplicate(b.right(),v,found);
  return false;
}
```

Die rekursive Variante traversiert den Baum und hält den aktuellen Suchstatus im dritten Parameter fest.

b)
```
boolean duplicate(BinaryTree b, int v)
{
  Stack firstFound, firstNotFound; // Keep track of found and other instances
  firstNotFound.push(b); // initialize stack
  bool found;
  BinaryTree current;
  while (!firstNotFound.isEmpty() || !firstFound.isEmpty()) // as long as there are nodes left
  {
    if (!firstFound.isEmpty){ // give priority to subtrees where the element has been found
      current = firstFound.pop();
      if (current.value() == v) return true;
      found = true;
    } else { // otherwise try to find first occurence
      current = firstNotFound.pop();
      found = (current.value() == v);
    }
    // push children to correct stack
    if (current.left() != null)
      if (found)
        firstFound.push(current.left());
      else
        firstNotFound.push(current.left());
    if (current.right() != null)
      if (found)
        firstFound.push(current.right());
      else
        firstNotFound.push(current.right());
  }
  return false;
}
```

Die iterative Variante behält 2 Stacks: Einen für die Knoten, in denen das Element noch nicht gefunden wurde, und einen für die Knoten, in denen das Element schon einmal gefunden wurde. Der letztere wird bevorzugt und die beinhalteten Teilbäume traversiert. Genau wie für den ersten Stack gilt dabei: Wenn Kinder vorhanden sind, werden diese auf den jeweils korrekten Stack geschoben.
Anschließend wird der nächste Knoten vom Stack geholt und überprüft.

Aufgabe 6 (ADT verwenden): (6 Punkte)

Gegeben sei ein Datentyp `Graph`, um ungerichtete Graphen zu repräsentieren. Dieser bietet die folgende Operation:

- `List<Node> getNeighbors(Node node)` gibt eine Liste, die alle Knoten des Graphen, die mit `node` durch eine Kante verbunden sind, enthält. Vorbedingung: `node` ist ein Knoten im Graph.

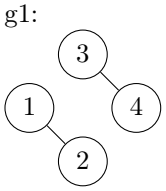
Dabei ist `List<Node>` der Datentyp für (unbegrenzte) Listen von Knoten, auf die mit den folgenden Methoden zugegriffen werden kann:

- `Element insert(Node n)` fügt ein Element mit Schlüssel (Inhalt) `n` am Ende der Liste ein.
- `void remove(Element x)` entfernt das Element `x` aus der Liste. Vorbedingung: `x` ist in der Liste.
- `Element successor(Element x)` gibt das Nachfolgerelement von `x` in der Liste zurück. Vorbedingung: `x` ist in der Liste.
- `int length()` gibt die Länge der Liste zurück.
- `bool contains(Node n)` gibt zurück, ob ein Element mit Schlüssel `n` in der Liste enthalten ist.
- `Element head()` gibt das erste Element in der Liste zurück, wenn die Liste nicht leer ist, und `null` sonst.

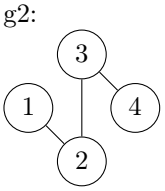
Im Graph `g` ist ein Knoten `n2` aus einem Knoten `n1` erreichbar, wenn es eine Sequenz $\mathtt{o}_1, \ldots, \mathtt{o}_k$ von Knoten im Graph `g` gibt, sodass $k \geq 1$, $\mathtt{o}_1 = \mathtt{n1}$, $\mathtt{o}_k = \mathtt{n2}$, und $\mathtt{o}_i$ und $\mathtt{o}_{i+1}$ durch eine Kante verbunden sind für alle $1 \leq i < k$.

Beispiel:

g1:



g2:




```
reach(g1,4,1} = false
reach(g1,4,3} = true
reach(g2,4,1} = true
reach(g2,4,4} = true
```

Entwerfen Sie einen Algorithmus `bool reach(Graph g, Node n1, Node n2)`, welcher überprüft, ob `n2` von `n1` in `g` erreichbar ist. Verwenden Sie dabei den DatenTyp `List<Node>`, um Knotenmengen abzuspeichern. Erklären Sie kurz den von Ihnen entworfenen Algorithmus textuell.

Lösung:

Eine mögliche Lösung bietet folgender Algorithmus:

```
bool reach(Graph g, Node n1, Node n2)
{
  if (n1 == n2) return true;
  List<Node> reach;
  Element current = reach.insert(n1); // Liste mit Knoten n1 enthalten
  while (current != null) {
    List<Node> neighbors = g.getNeighbors(current.key);
    Element e = neighbors.head();
    while (e != null)
    {
       if (e.key == n2) return true;
       if (!reach.contains(e.key)) reach.insert(e.key);
       e = neighbors.successor(e);
    }
    current = reach.successor(current);
  }
  return false;
}
```

Funktionsweise:
Ausgehend vom Startknoten `n1` wird eine Erreichbarkeitsmenge `reach` aufgebaut. Anfangs ist nur der Startknoten `n1` enthalten, da dieser offensichtlich von sich selber erreichbar ist. Nun werden für jeden Knoten in dieser Menge dessen Nachbarn hinzugefügt, da die Nachbarn eines erreichbaren Knotens auch erreichbar sind. Der Algorithmus terminiert mit der Rückgabe `true`, wenn der Zielknoten `n2` als Nachbar eines erreichbaren Knotens gefunden wird. Wenn ein *Fixpunkt* erreicht ist, in dem alle erreichbaren Zustände in `reach` enthalten sind, diese aber den Zielknoten `n2` nicht enthält, terminiert der Algorithmus mit der Rückgabe `false`.