

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 $\{e_1, \dots, e_n\}$ mit $e_i.key = i$ für alle $i \in \{1, \dots, n\}$. D. h. der Wertebereich enthält für alle $n \geq 1$ und $1 \leq k \leq n$ alle Mengen $\{T_1, \dots, T_k\}$ mit $T_i \subseteq \{e_1, \dots, e_n\}$, $e_i.key = i$ 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

$$\bigcup_{i=1}^k T_i = \{e_1, \dots, e_n\}.$$

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

Operationen:

- Der Konstruktor `UnionFind(int n)` erstellt die Partition $\{\{e_1\}, \dots, \{e_n\}\}$ von $\{e_1, \dots, e_n\}$ mit $e_i.key = i$ für alle $1 \leq i \leq n$. Vorbedingung: $n \geq 1$.
- `Element find(Element e)` gibt den Repräsentanten derjenigen Menge T_j aus der Partition zurück, die `e` enthält. Vorbedingung: $1 \leq e.key \leq n$.
- `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 `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`.

- 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.*
- 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);
    }
}

```

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 $I = \{-2^{31}, -2^{31} + 1, \dots, 2^{31} - 1\}$.

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.

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.

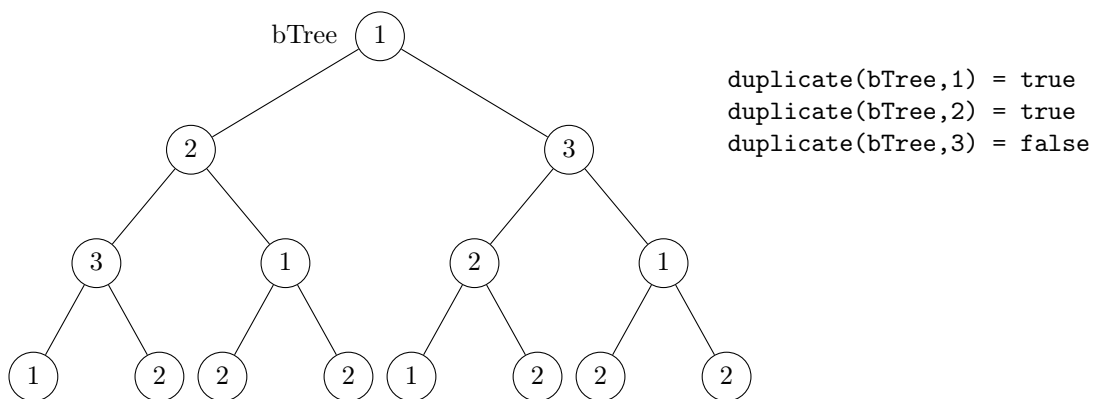
Ein Pfad in einem Binärbaum $b \neq \text{null}$ vom Typ `BinaryTree` ist eine Sequenz e_1, \dots, e_n von `BinaryTree` Objekten mit $n \geq 1$, $e_1 = b$, $e_{i+1} \in \{\text{left}(e_i), \text{right}(e_i)\}$ für alle $1 \leq i \leq n-1$ und $\text{left}(e_n) = \text{right}(e_n) = \text{null}$.

Entwerfen Sie

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



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 occurrence
      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.