Datenstrukturen
Übung – SS2020

Abgabe: 07.06. bis 24Uhr.
Abgabeort: Bitte laden Sie Ihre Klasse im OPAL-Bereich "Abgabe_Pflichtaufgabe_5" hoch. Ergänzen Sie bei Ihrer hochgeladenen Klasse die Namen (Vor- und Nachname) aller beteiligten Personen. Sie können Ihre Abgabe bis zur Abgabefrist beliebig oft löschen und erneut einreichen.

Wichtig: Da es sich um eine Prüfungsvorleistung handelt, ist es wichtig, dass jeder Teilnehmer Zugang zu allen notwendigen Informationen hat. Deswegen bitten wir Sie Fragen direkt ins Forum, in den Thread "Fragen zur Pflichtaufgabe 5" zu stellen.

Packen Sie Ihre Klasse in das Package "PVL5_$GroupX". Sollten Sie nicht in einer Gruppe arbeiten, wählen Sie den Namen statt "PVL5_$Nachname_$Vorname".
Ihre Klasse soll den Namen "PVL5_$GroupX" bzw. "PVL5_$Nachname_$Vorname" besitzen.

Für die Abgabe: Packen Sie all Ihre Klassen in ein .zip Archiv mit dem Namen "Set_GroupX.zip"
[Anmerkung: .rar ist kein .zip]

Prüfungsvorleistung 5 – Mengen

Mengen sind ein Grundlegendes Konzept in der Mathematik. In der Informatik benutzen man diese häufig für Zugehörigkeiten. In dieser Prüfungsvorleistung soll es darum gehen, dass Sie eine Mengen-Datenstruktur implementieren. Sie dürfen dabei nicht die in der Sprache gegeben Mengen Datenstrukturen verwenden. (Beispiele für Java: "Set", "HashSet", "TreeSet", … | Beispiele für Python: "Set", …). Es gibt verschiedene Ansätze für diese Datenstruktur.

Implementieren Sie das gegebene Interface "Set".
(In Python benötigen Sie kein Interface, doch halten Sie sich an die folgenden Methoden-Signaturen).

Sollte zu einem ein Übergabeparameter "null" sein, geben Sie "null" zurück.

Die folgenden Methoden finden Sie im Interface "Set", hier finden Sie eine Spezifikation für das Verhalten der Methoden, Beispiele folgen weiter unten. Es wird im Folgenden häufig von der "aktuellen Menge" gesprochen, damit sei das Objekt gemeint, welches aufgerufen wurde.

Java: public PVL5_$GroupX(int element)
Python: __init__(self, element: int)

Sie bekommen ein Element übergeben. Sie sollen nun eine Menge, mit nur diesem Element erstellen.

Java: public Set union(Set toInsert)
Python: def union(self, toInsert: Set) -> Set

Sie bekommen ein "Set" übergeben, führen Sie eine Vereinigung von "toInsert" in die aktuelle Menge ein. Speichern Sie das Ergebnis in der aktuellen Menge. Verändern Sie nicht " toInsert ". Ein Element sei gleich, wenn die Integer-Werte numerisch gleich sind. Geben Sie die aktuelle Menge zurück.

Java: public Set cut(Set toCutWith)
Python: def cut(self, toCutWith: Set) -> Set

Sie bekommen ein "Set" übergeben, führen Sie einen Schnitt von "toCutWith" mit der aktuellen Menge durch. Speichern Sie das Ergebnis in der aktuellen Menge. Verändern Sie nicht "toCutWith". Ein Element sei gleich, wenn die Integer-Werte numerisch gleich sind. Geben Sie die aktuelle Menge zurück.

Java: public boolean isSubsetOf(Set sampleSet)
Python: def isSubsetOf(self, sampleSet: Set) -> bool

Sie bekommen ein "Set" übergeben, überprüfen Sie, ob die aktuelle Menge eine Teilmenge von "sampleSet" ist. Verändern Sie nicht die aktuelle Menge oder "sampleSet". Eine Menge ist eine Teilemenge einer Anderen, wenn alle eigenen Elemente Teil der Anderen sind.

Java: public Set relativeComplementWith(Set sampleSet)
Python: def relativeComplementWith(self, sampleSet: Set) -> Set

Sie bekommen ein "Set" übergeben, bilden Sie das relative Komplement von der aktuellen Menge mit "sampleSet". Speichern Sie das Ergebnis in der aktuellen Menge. Verändern Sie nicht "sampleSet". Ein Element sei gleich, wenn die Integer-Werte numerisch gleich sind. Geben Sie die aktuelle Menge zurück. Das relative Komplement von einer Menge A mit B, wenn A nach der Operation kein Element von B enthält.

Java: public boolean equals(Set sampleSet)
Python: def equals(self, sampleSet: Set) -> boolean

Sie bekommen ein "Set" übergeben, überprüfen Sie ob die aktuelle Menge "sampleSet" übereinstimmt. Verändern Sie nicht "sampleSet". Zwei Mengen seien gleich, wenn sie beide Teilemengen voneinander sind.

Java: public Iterator<Set> iterator()
Python: def __iter__(self) -> Set , def __next__(self) -> Set

Implementieren Sie einen Iterator. Dieser soll es ermöglichen über die "next" Methode stückweise auf jedes Element zuzugreifen. Dabei soll der Iterator ein einelementiges "Set" halten, welches das Element enthält.

Java: public List<Integer> asIntList()
Python: def asIntList() -> List[int]

Implementieren Sie eine Methode. Diese soll eine Liste zurückgeben, in der alle Elemente der Menge enthalten sind (Reihenfolge ist egal). (für Python: "from typing import List")

Beispiele (java-pseudo Code):

```
PVL5_$GroupX set1 = new PVL5_$GroupX(1)
PVL5_$GroupX set2 = new PVL5_$GroupX(2)
PVL5_$GroupX set3 = new PVL5_$GroupX(3)
PVL5_$GroupX set4 = new PVL5_$GroupX(4)
PVL5_$GroupX set5 = new PVL5_$GroupX(5)
PVL5_$GroupX set6 = new PVL5_$GroupX(6)
set1.union(set2).union(set4).union(set5)
set3.union(set4).union(set5).union(set6)
//set1 = {‘1‘, ‘2‘, ‘4‘, ‘5‘}
//set3 = {‘3‘, ‘4‘, ‘5‘, ‘6‘}
```

Die folgenden Beispiele sind unabhängig voneinander zu betrachten, als Grundlage gilt der oben stehende Code:

1)
```
set1.union(set3)
//set1 = {‘1‘, ‘2‘, ‘3‘, ‘4‘, ‘5‘, ‘6‘}
```

2)
```
set1.cut(set3)
//set1 = {‘4‘, ‘5‘}
```

3)
```
set1.isSubsetOf(set3)  // false
set2.isSubsetOf(set1)  // true
set1.isSubsetOf(set1)  // true
```

4)
```
set1.relativeComplementWith(s3)
//set1 = {‘1‘, ‘2‘}
```

5)
```
set1.equals (set3)  // false
set2.equals (set1)  // false
set1.equals (set1)  // true
```

6)
```
Iterator<Set> iter = set1.iterator()
iter.next()     // {‘1‘}
iter.next()     // {‘2‘}
iter.next()     // {‘4‘}
iter.next()     // {‘5‘}
```