Modul
Datenstrukturen, Algorithmen und Programmierung 1

Probeklausur

***Hinweise zur Bearbeitung*:**

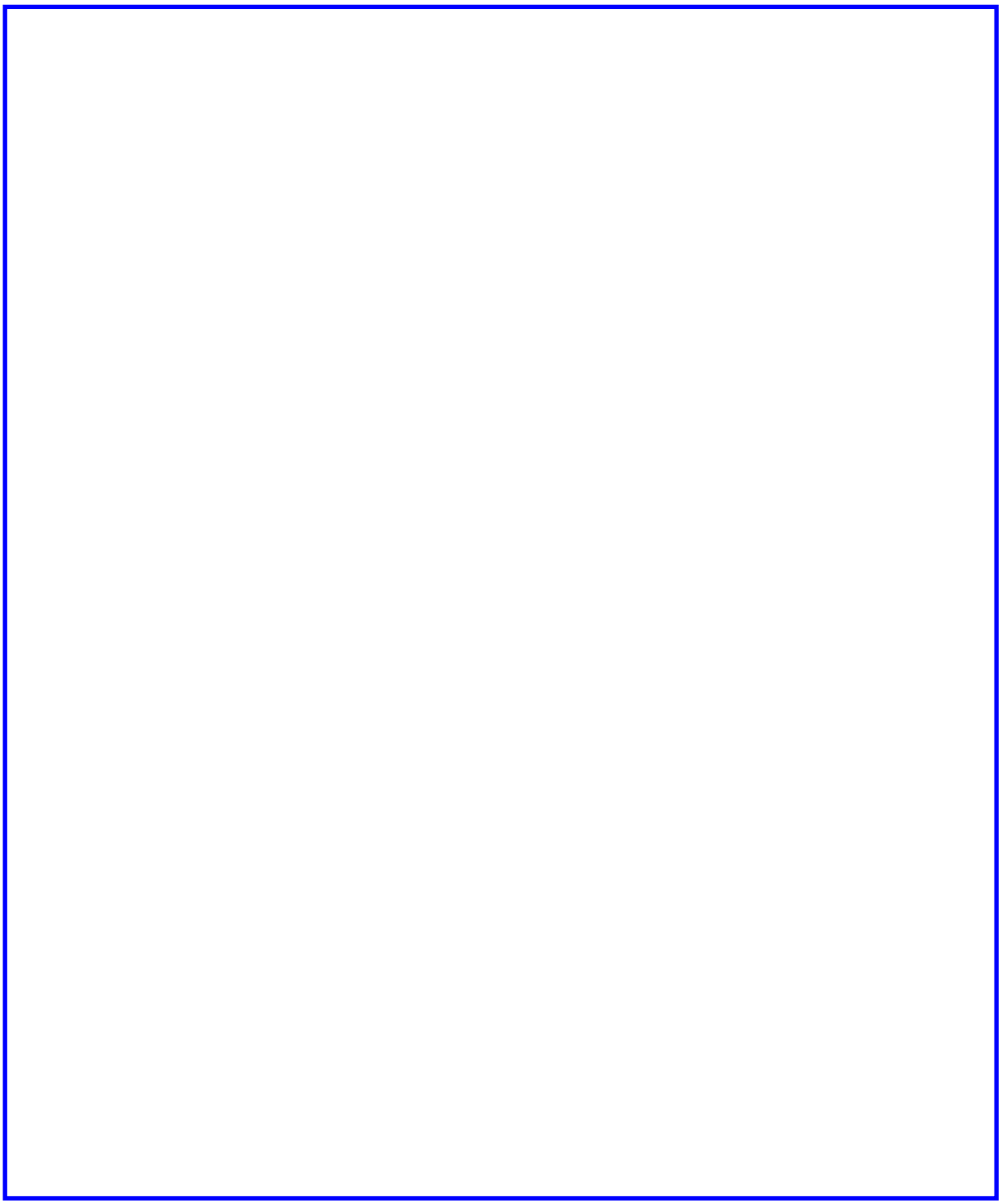
- Im Anhang dieser Klausur finden Sie die eventuell benötigten Programmtexte zu den vorgegebenen, bereits aus der Vorlesung bekannten Klassen und Interfaces. In den Lösungen zu den Aufgaben dieser Klausur dürfen aber nur die im Anhang aufgeführten Methoden benutzt werden.
- In dieser Klausur dürfen Sie immer nur vorgegebene Java-Programmtexte *mit Bezug* zu den durch Rahmen ☐ gekennzeichneten Stellen ergänzen
- An den vorgegebenen Programmtexten dürfen Sie *keine* Änderungen vornehmen.
- An *keiner* Stelle dieser Klausur dürfen Sie konzeptionell außerhalb von Rahmen ☐ liegende Programmtexte ergänzen.
- Sie müssen auf jeden Fall eine PDF-Datei abgeben, die alle Aufgaben dieser Klausur in der vorgegebenen Reihenfolge enthält.
- Idealerweise bearbeiten Sie die Klausur durch Ausfüllen und Sichern des vorgegebenen PDF-Formulars.
- Akzeptiert werden aber auch anders erstellte digitale Bearbeitungen oder digital erfasste Dateien von ausgedruckten oder handschriftlichen Klausurfassungen. Für den dadurch entstehenden Mehraufwand sind Sie selbst verantwortlich. Dieser muss innerhalb der Bearbeitungszeit geleistet werden.
- Gehen Sie bei allen Aufgaben davon aus, dass an einen Feld-Parameter immer ein Feld-Objekt (und nicht `null`) als Argument übergeben wird.

DAP 1 – Probeklausur

Aufgabe 1

Erstellen Sie eine Methode `public static int elemsBetween( int[] all )`.
Die Methode `elemsBetween` gibt die Anzahl der Elemente aus `all` zurück, die *zwischen* den beiden größten Werten in `all` liegen. Gehen Sie bei der Implementierung von `elemsBetween` davon aus, dass das Feld `all` mindestens die Länge 2 besitzt und in `all` *kein* Wert doppelt auftritt. Das Feld `all` muss bei der Ausführung der Methode `elemsBetween` unverändert bleiben.

```
public static int elemsBetween( int[] all )
{
}
```

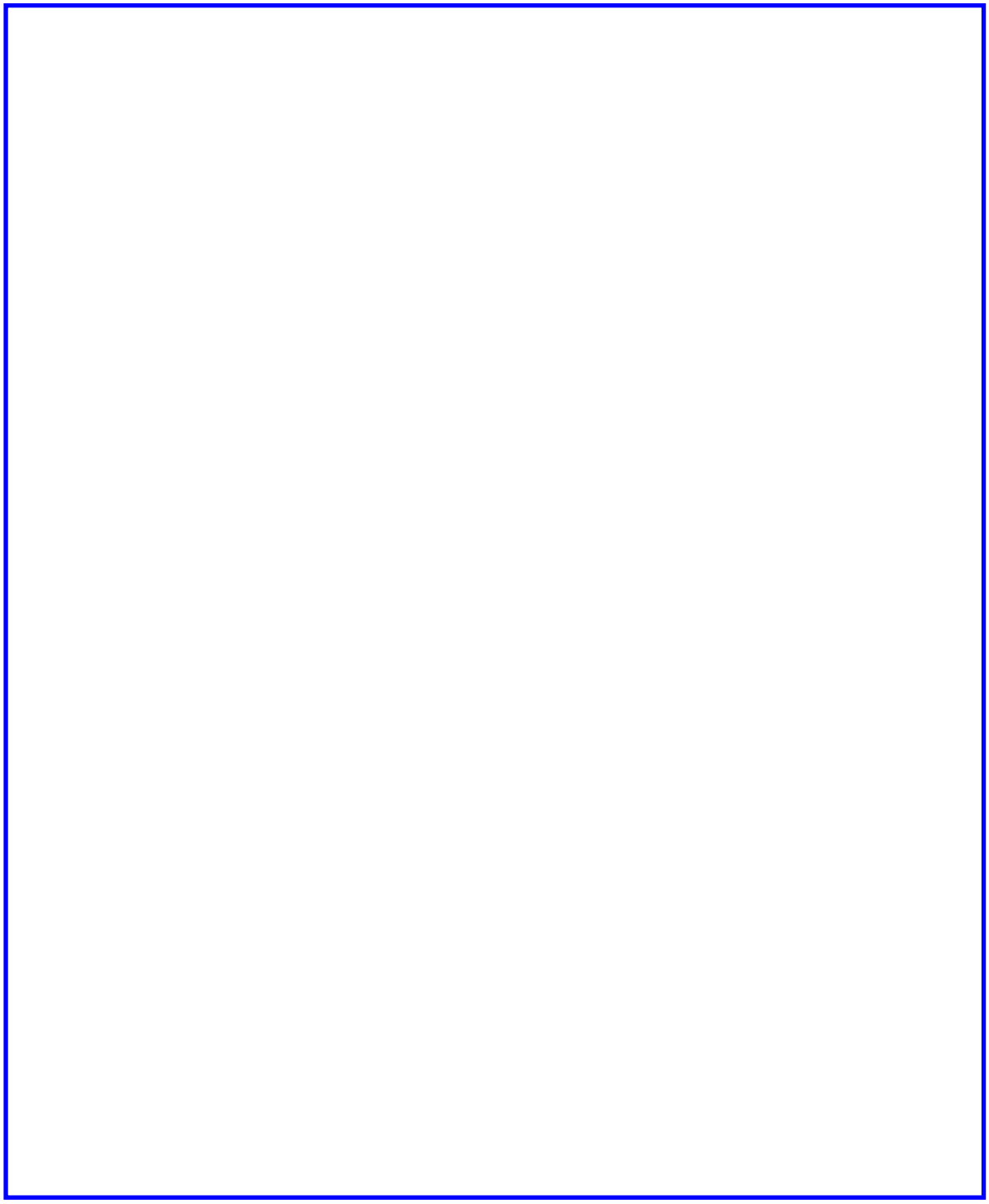


Aufgabe 2

Erstellen Sie eine Methode
`public static <T extends Comparable<T>> boolean hasSmaller( Iterable<T> all, T obj )`.
Die Methode `hasSmaller` gibt den Wert `true` zurück, falls im Feld `all` *genau ein* Objekt vorkommt, das kleiner als `obj` ist. Sonst soll `false` zurückgegeben werden.

Hinweis: Beachten Sie Folie 941 der Vorlesungsfolien.

```
public static <T extends Comparable<T>> boolean hasSmaller( Iterable<T> all, T obj )
{
}
```



Aufgabe 3

Gegeben ist die folgende Klasse `Parts`.

```
public class Parts<T> implements Iterable<T> {
    private Iterable<T> part1;
    private Iterable<T> part2;
    public Parts( Iterable<T> p1, Iterable<T> p2 ) {
        part1 = p1;
        part2 = p2;
    }
    public java.util.Iterator<T> iterator() {
        return new PartsIterator();
    }
}
```

Geben Sie eine Implementierung für die innere Klasse `PartsIterator` an. Der so realisierte Iterator soll zunächst alle Inhalte von `part1` und anschließend alle Inhalte von `part2` liefern.

```
private class PartsIterator implements java.util.Iterator<T>
{
    
    public boolean hasNext()
    {
        
    }
    public T next()
    {
        
    }
}
```

Aufgabe 4

Ergänzen Sie die aus der Vorlesung bekannte Klasse `DoublyLinkedList<T>`, die Sie im Anhang finden. Bei der Implementierung der geforderten Methode dürfen **nur die im Anhang** aufgeführten Methoden genutzt werden. Die Klasse `DoublyLinkedList<T>` soll um *genau eine* Methode ergänzt werden.

Erstellen Sie die Methode `public boolean cutRange( int pos1, int pos2 )`.

Die Methode `cutRange` löscht alle Elemente aus der ausführenden Liste, die im Bereich der Positionen von *einschließlich* `pos1` bis *einschließlich* `pos2` liegen, falls folgende drei Bedingungen gelten:

`0<=pos1<size()`, `0<=pos2<size()`, `pos1<pos2`.

Gilt eine dieser Bedingungen nicht, bleibt die Liste unverändert.

Das erste Element der Liste liegt an der Position `0`.

Wird mindestens ein Element gelöscht, gibt die Methode `cutRange` den Wert `true` zurück, sonst den Wert `false`.

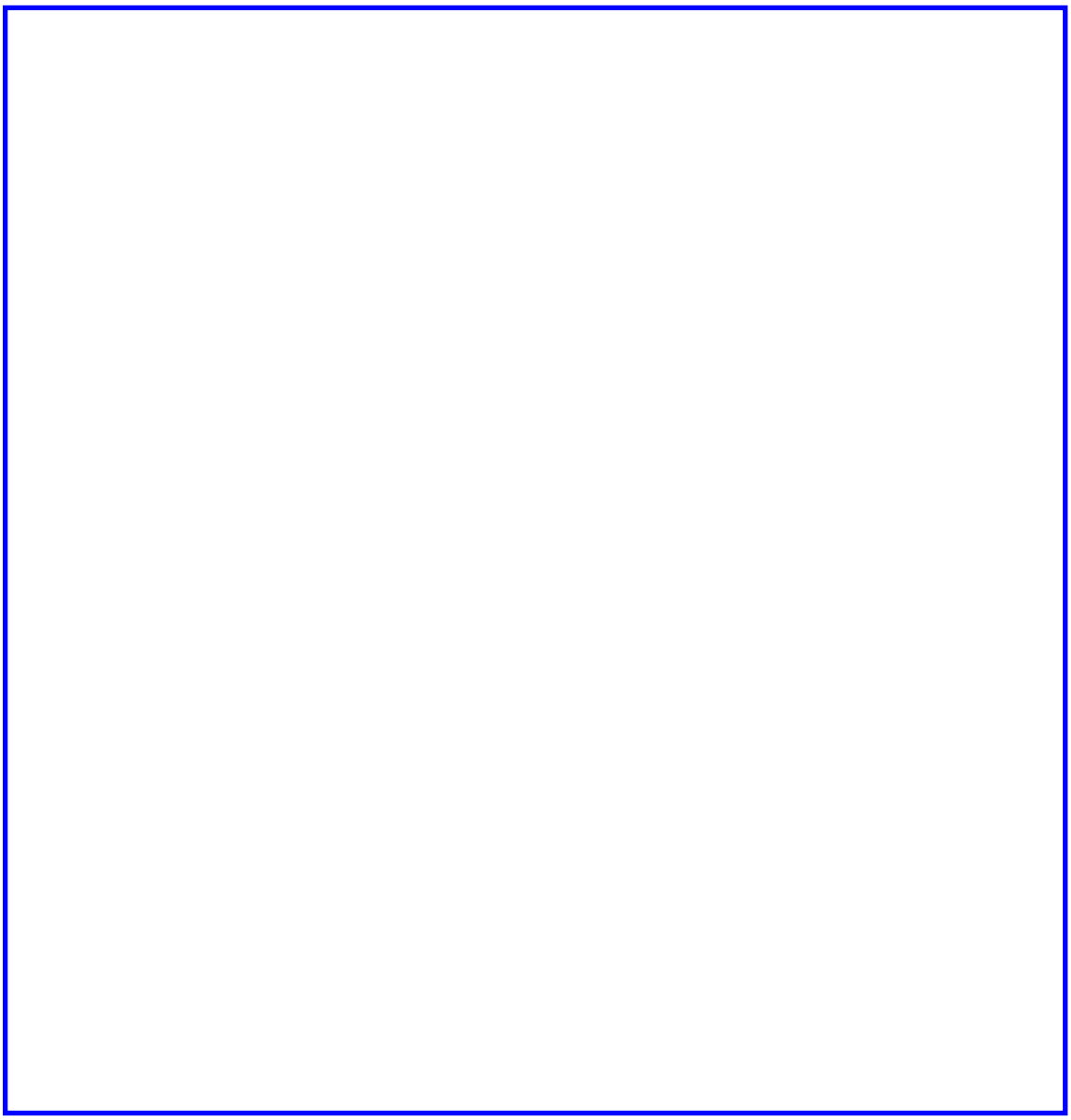
```
public boolean cutRange( int pos1, int pos2 )
{
    if ( pos1 >=0 && pos1 < size && pos2 >=0 && pos2 < size ) {
        if ( pos1 > pos2 ) {
            int posH = pos1; pos1 = pos2; pos2 = posH;
        }
        int count = 0;
        Element delFirst;
        Element delLast;
        Element current = first;
        while ( count != pos1 ) {
            count++;
            current = current.getSucc();
        }
        delFirst = current;
        while ( count != pos2 ) {
            count++;
            current = current.getSsucc();
        }
        
    }
    return false;
}
```

Aufgabe 5

Ergänzen Sie die aus der Vorlesung bekannte Klasse `BinarySearchTree<T extends Comparable<T>>`, die Sie im Anhang finden. Bei der Implementierung der geforderte Methode dürfen **nur die im Anhang** aufgeführten Methoden genutzt werden. Die Klasse `BinarySearchTree` soll um *genau eine* Methode ergänzt werden.

Erstellen Sie die Methode `public void allInner( java.util.List<T> collect )`.

Die Methode `allInner` ergänzt die als Argument an den Parameter `collect` übergebene Liste um alle Inhalte des Baums, die *nicht* in Blättern abgelegt sind. Diese Inhalte sollen in absteigender Reihenfolge zu `collect` hinzugefügt werden. Der Baum darf durch die Ausführung der Methode `allInner` nicht verändert werden.

```
public void allInner( java.util.List<T> collect )
{
    if ( !isEmpty() )
    {
        
    }
}
```

Aufgabe 6

Ergänzen Sie die aus der Vorlesung bekannte Klasse `BinarySearchTree<T extends Comparable<T>>`, die Sie im Anhang finden. Bei der Implementierung der geforderte Methode dürfen **nur die im Anhang** aufgeführten Methoden genutzt werden. Die Klasse `BinarySearchTree` soll um *genau eine* Methode ergänzt werden.

Vorgegeben ist die Methode `onPathTo`, die den Wert `true` zurückgibt, falls ein Knoten mit dem Inhalt `target` existiert und auf dem Pfad von der Wurzel zu diesem Knoten ein Knoten mit dem Inhalt `obj` liegt. Sonst wird der Wert `false` zurückgegeben.

```
public boolean onPathTo( T obj, T target ) {
    return pathCheck( obj, target, false );
}
```

Erstellen Sie die Methode `private boolean pathCheck( T obj, T target, boolean found )` derart, dass die Methode `onPathTo` die oben beschriebene Funktionalität bietet. Den Parameter `found` können Sie dabei geeignet einsetzen.

Der Baum darf durch die Ausführung der Methode `pathCheck` nicht verändert werden.

```
private boolean pathCheck( T obj, T target, boolean found )
{
    if (!isEmpty() )
    {
        
    }
    else
    {
        return false;
    }
}
```

Aufgabe 7

Gegeben sind das Interface `IntFunction` und die Klasse `Data`.

```
public interface IntFunction {
    int apply( int x, int y );
}
```

```
public class Data {
    private int[] intValues;
    public Data( int[] iV ) {
        intValues = iV;
    }
    public int doInt( IntFunction f ) {
        int result = 0;
        for ( int v : intValues ) {
            result = f.apply( v, result );
        }
        return result;
    }
}
```

a) Erstellen Sie eine Methode `static boolean moreThan( Data d, int i1, int i2 )`.

Die Methode `moreThan` gibt den Wert `true` zurück, falls `intValues` den Wert `i1` häufiger als den Wert `i2` enthält. Sonst wird `false` zurückgegeben.

```
static boolean moreThan( Data d, int i1, int i2 ) {

}
```

b) Erstellen Sie eine Methode `static boolean mostAreSmaller( Data d, int p )`.

Die Methode `mostAreSmaller` gibt den Wert `true` zurück, falls mehr als die Hälfte aller in `intValues` enthaltenen Werte kleiner als `p` sind. Sonst wird `false` zurückgegeben.

```
static boolean mostAreSmaller( Data d, int p ) {

}
```

c) Erstellen Sie eine Methode `static int addIf( Data d, int limit )`.

Die Methode `addIf` gibt die Summe aller Werte aus `intValues` zurück, falls die Länge von `intValues` kleiner als `limit` ist. Sonst wird `0` zurückgegeben.

```
static int addIf( Data d, int limit ) {

}
```

DAP 1 – Probeklausur

Anhang – Programmcode der Klasse `BinarySearchTree<T extends Comparable<T>>`

```
public class BinarySearchTree<T extends Comparable<T>> {
    private T content;
    private BinarySearchTree<T> leftChild, rightChild;
    public BinarySearchTree() { ... }
    public T getContent() { ... }
    public boolean isEmpty() { ... }
    public boolean isLeaf() { ... }
}
```

Anhang – Programmcode der Klasse `DoublyLinkedList<T>`

```
public class DoublyLinkedList<T> {
    private Element first, last;
    private int size;
    public DoublyLinkedList() { ... }
    public int size() { ... }
    public boolean isEmpty() { ... }

    // Element
    private static class Element {
        private T content;
        private Element pred, succ;
        public Element( T c ) { ... }
        public T getContent() { ... }
        public void setContent( T c ) { ... }
        public boolean hasSucc() { ... }
        public Element getSucc() { ... }
        public void connectAsSucc( Element e) { ... }
        public void disconnectSucc() { ... }
        public boolean hasPred() { ... }
        public Element getPred() { ... }
        public void connectAsPred( Element e ) { ... }
        public void disconnectPred() { ... }
    }
}
```

Anhang – Programmcodes der Interfaces

```
public interface Iterator<T> {
    public abstract boolean hasNext();
    public abstract T next();
}


public interface Iterable<T> {
    public abstract Iterator<T> iterator();
}


public interface Comparable<T> {
    public abstract int compareTo( T t );          // Der Rückgabewert ist positiv, falls das
}                                                  // ausführende Objekt größer als t ist.
```

DAP 1 – Probeklausur