Prof. Dr. Dirk Riehle, M.B.A.
Prof. Dr. Michael Philippsen
Department Informatik, FAU

30. März 2015

Klausur: Algorithmen und Datenstrukturen

Angaben zur Person (Bitte Etikett aufkleben bzw. in Druckbuchstaben ausfüllen!):

Name, Vorname: .....

Laufende Nr.: .....

Matrikelnummer:

.....

Bitte kleben Sie hier das Etikett auf.

Folgende Hinweise bitte lesen und Kenntnisnahme durch Unterschrift bestätigen!

- **Hilfsmittel außer Schreibmaterialien sind *nicht* zugelassen**.
- Lösungen müssen in den dafür vorgesehenen freien Raum geschrieben werden. Sollte der Platz nicht ausreichen, verwenden Sie zunächst (mit kurzem Hinweis) die Zusatzseite am Ende. Weitere Zusatzseiten müssen von der Aufsicht ausgegeben und *eingeheftet* werden.
- Sie können Schmierpapier von der Aufsicht anfordern. ***Das Schmierpapier darf nicht mit abgegeben werden.***
- Können Sie die Prüfung aus gesundheitlichen Gründen nicht fortsetzen, dann müssen Sie Ihre Prüfungsunfähigkeit durch Vorlage eines *erweiterten ärztlichen Attestes* beim Prüfungsamt nachweisen. Melden Sie sich in jedem Fall bei der Aufsicht und lassen Sie sich das *entsprechende Formular* aushändigen.
- ***Überprüfen Sie diese Klausur auf Vollständigkeit (16 Seiten inkl. Deckblatt) und einwandfreies Druckbild!***

Durch meine Unterschrift bestätige ich den Empfang der vollständigen Klausurunterlagen und die Kenntnisnahme der obigen Informationen.

Erlangen, den 30.03.2015 .....
(Unterschrift)

Nicht von der Kandidatin bzw. vom Kandidaten auszufüllen!

Bewertung (Punkteverteilung unter Vorbehalt):

Aufgabe	1	2	3	4	5	6	7	$\sum$
Maximal	16	12	10	18	20	15	29	120
Erreicht	3	8	9					20

Aufgabe 1 (Wissensfragen) (16 Punkte)

Bei den folgenden Teilaufgaben werden richtige Kreuze positiv (+) und falsche **oder fehlende** Kreuze entsprechend negativ (–) gewertet. Jede Teilaufgabe wird mit mindestens 0 Punkten bewertet. Pro Teilaufgabe ist mind. eine Aussage wahr. Kreuzen Sie alle richtigen Aussagen an:

a) Gegeben sei ein Binärbaum mit Wurzel n.

- [x] Falls bei einem ***pre**-order*-Besuch des Baumes alle Knoten in **auf**steigender Reihenfolge vorkommen, ...
- [ ] Falls bei einem ***post**-order*-Besuch des Baumes alle Knoten in **ab**steigender Reihenfolge vorkommen, ...
- [ ] Falls alle Knoten in dessen **linken** Unterbaum **kleiner** und im rechten größer als n sind, ...
- [ ] Falls alle Knoten in dessen **linken** Unterbaum **größer** und im rechten kleiner als n sind, ...

dann ist die Min-Heap-Eigenschaft erfüllt.

b) Deklariert man das Attribut `private static final int[] A = new int[42];` in einer Java-Klasse namens `JClass`, dann kann man ...

- [x] der Variablen `A` kein neues Feld mehr zuweisen, z.B. mit `A = new int[666];`
- [ ] einzelne Einträge im Feld `A` nicht mehr ändern, z.B. mit `A[0] = 42;`
- [ ] aus Klassen- und Instanzmethoden in `JClass` lesend auf `A` zugreifen.
- [x] nur aus Klassenmethoden in `JClass` lesend auf `A` zugreifen.

c) Sei $G = (V, E)$ ein zusammenhängender Graph in Adjazenzlistendarstellung mit $v \in V$ und $n := |V|$ bzw. $e := |E|$. Welche Laufzeitkomplexität hat die Methode `bfs`?

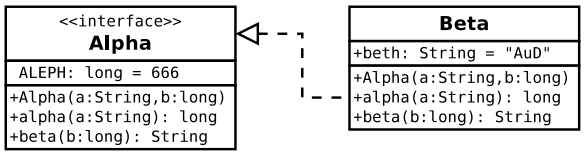
```
void bfs(Node v) { // Breitensuche
    Queue<Node> q = new LinkedList<>(); // ∈ O(1)
    q.add(v); // ∈ O(1)
    while (!q.isEmpty()) { // isEmpty ∈ O(1)
        v = q.poll(); // ∈ O(1)
        v.mark(); // ∈ O(1)
        Iterator<Edge> iter = v.getEdges(); // ∈ O(1)
        while (iter.hasNext()) { // hasNext ∈ O(1)
            Edge e = iter.next(); // ∈ O(1)
            if (e.target.isUnmarked()) { // ∈ O(1)
                q.add(e.target); // ∈ O(1)
            }
        }
    }
}
```

- [ ] $\mathcal{O}(1)$
- [ ] $\mathcal{O}(log_2(n+e))$
- [ ] $\mathcal{O}(\sqrt{n+e})$
- [x] $\mathcal{O}(n+e)$

d) Für einen Graphen mit n Knoten und e Kanten ...

- [ ] liegt der Speicherplatzbedarf für die Adjazenz**matrix** in $\mathcal{O}(n \cdot e)$.
- [ ] findet man alle k Nachfolger eines Knotens in der Adjazenz**matrix** in $\mathcal{O}(n)$ Zeit.
- [x] liegt der Speicherplatzbedarf für die Adjazenz**liste** in $\mathcal{O}(log_2(n))$.
- [x] findet man alle k Nachfolger eines Knotens in der Adjazenz**liste** in $\mathcal{O}(k)$ Zeit.

e) Welche Code-Zeilen dürfen in der Implementierung der Klasse `Beta` vorkommen, die folgendem UML-Diagramm gehorcht?

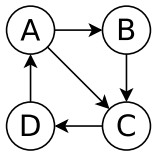


- [ ] `super();`
- [ ] `super("AuD", 666);`
- [x] `return this.beta(Alpha.ALEPH);`
- [ ] `return super.alpha(this.beth);`

f) Welche der folgenden Aussagen ist/sind korrekt?

- [ ] Tritt in einem `catch`-Block eine Ausnahme auf, dann wird der zugehörige `finally`-Block **nicht** mehr ausgeführt.
- [x] Ein `try` kann mehrere `catch`-Blöcke für verschiedene Ausnahmetypen haben.
- [ ] Im `finally`-Block darf **keine** weitere `Exception` auftreten/geworfen werden.
- [x] Ein mit `catch` gefangenes Objekt kann mit `throw` weitergeworfen werden, z.B.: `try { /*...*/ } catch (Exception e) {throw e;}`

g) Welche Aussagen treffen auf folgenden gerichteten Graphen zu?



- [ ] Der Pfad A, B, C, D, A ist ein sogenannter *minimaler Zyklus*.
- [x] A ist ein sogenannter *Vorgänger* (oder *Vorfahr*) von D.
- [ ] Ohne Kante (B, C) wäre der Graph ein sogenannter DAG.
- [x] Lässt man die Kante (B, C) weg und betrachtet alle Kanten als ungerichtet, dann ergibt sich ein *Spannbaum* des Graphen.

h) Wendet man die in der Vorlesung gezeigten Implementierungen der Sortierverfahren auf ein Feld mit n Elementen an, dann ...

- [ ] hat Sortieren durch Verschmelzen eine Worst-Case-Laufzeit von $\mathcal{O}(n \cdot log(n))$.
- [ ] hat Sortieren durch Einfügen eine Average-Case-Laufzeit von $\mathcal{O}(n \cdot log(n))$.
- [x] arbeitet die Haldensortierung *stabil* **und** *in situ*.
- [x] arbeitet die Blasensortierung *stabil* **und** *in situ*.

Aufgabe 2 (Sortieren durch Zählen) (12 Punkte)

Gegeben sei das `char`-Feld `in`, dessen Einträge garantiert im Wertebereich 0 bis 255 liegen. Dieses Feld soll in zwei Schritten aufsteigend sortiert werden. Im ersten Schritt (Teilaufgaben a und b) wird gezählt, wie häufig jeder mögliche Buchstabe tatsächlich auftritt. Im zweiten Schritt (Teilaufgabe c) wird das sortierte Ergebnis anhand der ermittelten Häufigkeiten erzeugt.

```
public class CountingSort {
    public static void sort(char[] in) {
```

a) Initialisieren Sie zunächst das Feld `count` zum Erfassen der Häufigkeiten.

```
int[] count = new int[256];
```

b) Durchlaufen Sie das Feld `in` und zählen Sie die Häufigkeiten der enthaltenen Buchstaben.

```
for(int i=0; i< count.length(); i++){
    count[in[i]]++;
}
```

c) Die Anzahl der Vorkommen von Buchstabe `i` in der Eingabe `in` steht nun in `count[i]`. Vervollständigen Sie die Schleife und füllen Sie das Feld `in` mit den sortierten Buchstaben.

```
char c = 0; //current char
int index = 0; //write index
do {
    if(count[c]>0){
        in[index] = c;

        index++;

    }
    c++;
} while ( c < count.length() );
```

d) Welche worst-case-Laufzeit hat das obige Verfahren zum Sortieren von n Elementen?

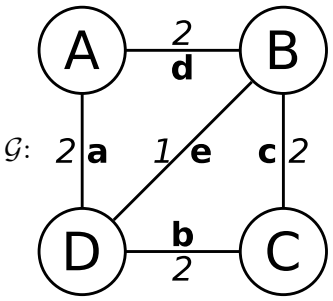
$\mathcal{O}(\ n\ )$ linear in n

e) Ist die bestmögliche worst-case-Laufzeit $\mathcal{O}(n \cdot \log(n))$ für vergleichsbasiertes Sortieren damit widerlegt?

☐ ja ☒ nein

Aufgabe 3 (Prim vs. Kruskal) (10 Punkte)

Gegeben sei der gewichtete Graph $\mathcal{G}$ und seine unterschiedlich sortierten Kanten:



lexikographisch für *Prim*

Kante	*Name*	*Gewicht*
(A,B)	d	2
(A,D)	a	2
(B,C)	c	2
(B,D)	e	1
(C,D)	b	2

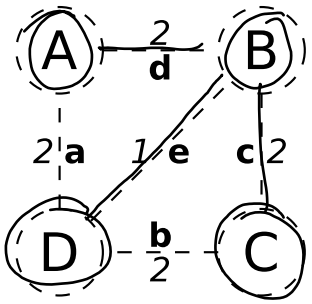
alphabetisch für *Kruskal*

Name	*Kante*	*Gewicht*
a	(A,D)	2
b	(C,D)	2
c	(B,C)	2
d	(A,B)	2
e	(B,D)	1

a) Geben Sie die Kanten von $\mathcal{G}$ in der Reihenfolge an, in welcher der Algorithmus von *Prim* sie in den Spannbaum aufnimmt, falls er beim Knoten A startet und Konflikte dadurch löst, dass er die lexikographisch kleinere Kante wählt, also z.B. (X, Y) *vor* (X, Z), falls er unter mehreren Kanten mit gleichem Gewicht wählen kann:

d, e, c

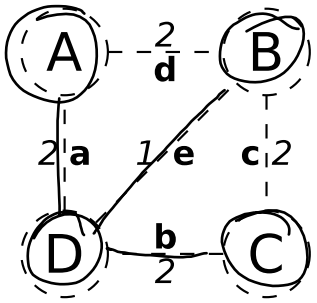
Schmiergraph:
*(wird **nicht** bewertet)*



b) Geben Sie die Kanten von $\mathcal{G}$ in der Reihenfolge an, in welcher der Algorithmus von *Kruskal* sie in den Spannbaum aufnimmt, falls er Konflikte dadurch löst, dass er die alphabetisch kleinere Kante wählt, also z.B. x *vor* y, falls er unter mehreren Kanten mit gleichem Gewicht wählen kann:

e, a, b

Schmiergraph:
*(wird **nicht** bewertet)*



c) Wenn *alle* Kanten des Graphen *unterschiedliche* Gewichte hätten, kann es trotzdem vorkommen, dass *Prim* und *Kruskal* verschiedene Spannbäume ermitteln?

☐ ja ☒ nein – ∧

Aufgabe 4 (Streuspeicherung) (18 Punkte)

a) Gegeben seien die folgenden Schlüssel k zusammen mit ihren Streuwerten $h(k)$:

k	A	B	C	D	E	F	G
$h(k)$	1	3	3	3	2	3	3

Fügen Sie die Schlüssel k in alphabetischer Reihenfolge mit der Streufunktion $h(k)$ in die folgende Streutabelle ein. Verwenden Sie ~~*geschlossenes*~~ Hashing mit **Behältergröße b=2** und lösen Sie Kollisionen durch **lineares Sondieren** mit Schrittweite **c=+2** auf.

	Bucket ▶	0	1	2	3	4
▼	*erster Schlüssel:*	D	A	E	B	
%	*zweiter Schlüssel:*	F		G	C	

b) Ihre Methode `put` soll den Schlüssel `k` mit Streuwert `hk` in die Tabelle `map` einfügen. Die `s` Behälter der Tabelle nehmen jeweils bis zu `b` Schlüssel auf und weitere Kollisionen werden durch lineares Sondieren mit Schrittweite `c` gelöst. Findet `put` auf der Suche nach einem freien Platz zuerst einen Schlüssel k_{old}, der mit `k` bzgl. `equals` gleich ist, ersetzt sie k_{old} durch `k` und gibt das alte Objekt k_{old} zurück, andernfalls gibt sie `null` zurück. Gerät das Sondieren in einen *Zyklus*, so muss `put` eine `IllegalArgumentException` werfen.

```
class HashSet<K> {
    K[][] map;
    int s, b, c;

    HashSet(int s, int b, int c) {
        assert 0 < c && c < s;
        this.s = s; // size of map
        this.b = b; // bucket size
        this.c = c; // collision increment
        map = (K[][]) new Object[s][b];
    }
```

```
K put(K k, int hk) {
    assert k != null && 0 <= hk && hk < s;
    int pos = hk; // current position during exploration
    do {
```

Aufgabe 5 (Dynamische Programmierung) (20 Punkte)

a) Implementieren Sie `b(n,m)` ***rekursiv*** gemäß folgender Definition:

$$b(n,m) := \begin{cases} \frac{3\cdot(n-1)\cdot b(n-1,m)+m\cdot b(n-1,m-1)}{n} & falls \quad n-1 \geq m \geq 1 \\ 0 & falls \quad (n < m) \vee (m = 0 \wedge n \neq 0) \\ 1 & falls \quad n = m \\ \texttt{IllegalArgumentException} & sonst \end{cases}$$

```
long b(int n, int m) {
    if (
        return
    } else if (
        return 0;
    } else if (
        return 1;
    }
}
```

b) Die Methode `b(n,m)` aus Teilaufgabe (a) ruft sich rekursiv sowohl mit `b(n-1,m)` als auch mit `b(n-1,m-1)` auf. Schneller als diese Implementierung ist eine Methode mit dynamischer Programmierung, bei der die Zwischenergebnisse `b(i,j)` für $0 \leq i \leq n \wedge 0 \leq j \leq m$ ***iterativ*** berechnet und in einem Feld aufbewahrt werden. Ergänzen Sie eine solche Variante in der folgenden Methode `bDP(n,m)`. Zur Vereinfachung wird Ihnen hier zugesichert, dass diese Methode von außen nur mit gültigen Parametern ($n \geq m \geq 1$) aufgerufen wird.

```
long bDP(int n, int m) {
    long[][] mem = new long[n + 1][m + 1];
```

```
    for (int i =
```

```
        for (int j =
```

Aufgabe 6 (Abstrakte Datentypen) (15 Punkte)

Gegeben sei der ADT *Nat* zur Repräsentation natürlicher Zahlen $\mathbb{N}_0^+$:

adt *Nat*
sorts *Nat*
ops

null:		$\mapsto$ *Nat*	// *entspricht* `null` *in Java*	
zero:		$\mapsto$ *Nat*	// *entspricht der Zahl „0“*	
succ:	*Nat*	$\mapsto$ *Nat*	// *Nachfolger „$(x+1)$“ der Nat-Zahl x*	
add:	*Nat* $\times$ *Nat*	$\mapsto$ *Nat*	// *Addition im Nat-Raum*	
mul:	*Nat* $\times$ *Nat*	$\mapsto$ *Nat*	// *Multiplikation im Nat-Raum*	
div:	*Nat* $\times$ *Nat*	$\mapsto$ *Nat*	// *ganzzahlige Division im Nat-Raum (immer abgerundet)*	
mod:	*Nat* $\times$ *Nat*	$\mapsto$ *Nat*	// *Rest bei Nat-Division (modulo, in Java: %)*	

axs
... // *aus Platzgründen weggelassen*
end *Nat*

Betrachten Sie nun das folgende Gerüst des ADTs *NatStack* (Stapel mit *Nat*-Elementen):

adt *NatStack*
sorts *NatStack, Nat*
ops

empty:		$\mapsto$ *NatStack*	// *erzeugt einen neuen leeren Stapel*
push:	*NatStack* $\times$ *Nat*	$\mapsto$ *NatStack*	// *legt ein Nat* ***auf*** *den Stapel*
peek:	*NatStack*	$\mapsto$ *Nat*	// *gibt bei leerem Stapel* `null` *und bei* // *nicht-leerem Stapel das* ***„oberste“*** *Nat zurück*
pop:	*NatStack*	$\mapsto$ *NatStack*	// *gibt bei leerem Stapel empty und bei nicht-leerem*
...			// *Stapel den Stapel ohne das* ***„oberste“*** *Nat zurück*

axs
...
end *NatStack*

a) Ergänzen Sie *NatStack* um diejenigen Axiome, die das Zusammenspiel von *peek* bzw. *pop* mit den Primärkonstruktoren *empty* und *push* spezifizieren:
axs

b) Ergänzen Sie den ADT *NatStack* um Axiome, die das Verhalten der folgenden Methode `nat2bin` spezifizieren, die für eine *Nat*-Zahl ihre Binärdarstellung (Zweierkomplement) als *NatStack* erzeugt:

```
public static NatStack nat2bin(Nat n) {
    if (n == zero)
        return push(empty, zero);
    else
        return push(nat2bin(div(n, succ(succ(zero)))),
                mod(n, succ(succ(zero))));
}
```

Beispiel: $42_{(10)} \overset{nat2bin}{\longmapsto} \left\{ \begin{array}{c} \underline{zero} \\ succ(zero) \\ zero \\ succ(zero) \\ zero \\ succ(zero) \\ zero \end{array} \right\} = 0101010_{(2)}$

ops

nat2bin: $Nat \rightarrow NatStack$

axs

c) Ergänzen Sie den ADT *NatStack* um Axiome für die Operation `bin2nat`, die aus einer Binärdarstellung im Stapel die zugehörige *Nat*-Zahl berechnet. Es wird garantiert, dass nur *zero* oder *succ*(*zero*) auf dem Stapel liegen und dass das *niederwertigste* Bit das **„oberste“** ist:

ops

bin2nat: $NatStack \rightarrow Nat$ *// Umkehrfunktion von nat2bin*

axs

bin2nat(empty) =

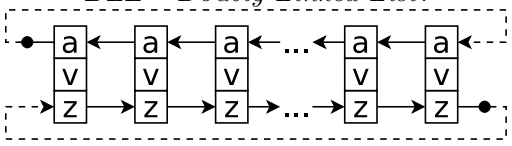
bin2nat(push(s, n)) =

Aufgabe 7 (Doppelverkettung) (29 Punkte)

Gegeben sei folgende Java-Klasse, wie sie z.B. für doppelt verkettete Listen verwendet wird:

```
class DLNode<V> {
    DLNode<V> a;
    V v;
    DLNode<V> z;
    // ...
```

DLL – *Doubly Linked List*:



Gestrichelte Kanten:
DLL *kann* ggf. auch zirkulär verkettet sein!

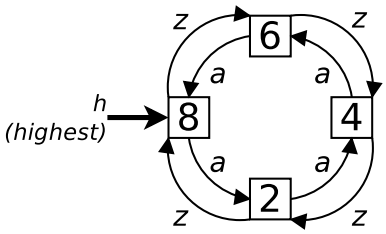
a) Ergänzen Sie die Methode `isDLL`, die genau dann `true` zurückgibt, wenn der aktuelle Knoten `this` und die *von ihm aus erreichbaren* Knoten eine doppelt verkette Liste bilden:

```
boolean isDLL() { // checks if this node is part of a Doubly Linked List
    DLNode<V> d = this, c = this.a; // drag/current pointers
```

b) Die Methode `isLoopedDLL` gibt genau dann `true` zurück, wenn der aktuelle Knoten `this` und die von ihm aus erreichbaren Knoten eine doppelt verkettete **und *zirkuläre*** Liste bilden. Verwenden Sie die Methode `isDLL` aus Teilaufgabe (a).

```
boolean isLoopedDLL() { // checks if this node is part of a Looped DLL
    DLNode<V> c = this.a;
```

Die folgende `PriorityDeque` verwaltet ihre Werte nach Priorität sortiert (hier definiert durch die `Comparable`-Ordnung) in einer zirkulären doppelt verketteten Liste, indem sie im Attribut `h` stets den Knoten mit der höchsten Priorität speichert.




```
class PriorityDeque<V extends Comparable<V>> {
    DLNode<V> h; // highest prio node (null if empty)
```

c) Ergänzen Sie die Methode `get`: Wird sie mit `true` (bzw. `false`) aufgerufen, dann soll sie den Knoten mit höchster (bzw. niedrigster) Priorität aus der Liste entfernen und seinen Wert zurückgeben. Ist die Deque leer, soll `get` eine `NoSuchElementException` werfen.

```
V get(boolean high) {
    DLNode<V> r = h; // will finally point to requested node
    // list is empty
    if (h == null) {
```

```
    // list has only one single node - just empty the deque
```

```
    } else if (
```

```
    // user requested head - reconfigure internal head pointer
```

```
    } else if (high) {
```

```
    // user requested node that preceeds head - retarget r
```

```
    } else {
```

```
    }
    // unlink requested node r and return its value
```

d) Ergänzen Sie die Methode `put`, die `v` in die `PriorityDeque` aufnimmt und dabei ggf. auch das Attribut `h` anpasst:

```
void put(V v) {
    DLNode<V> c = h; // current node (drag)
    DLNode<V> n = new DLNode<V>(); n.v = v; n.a = n.z = n; // new node
    if (c == null || c.v.compareTo(v) < 0) {
        // n becomes new head
```

```
    } else {
        // search node c that immediately succeeds n
        do {
```

```
        } while (
    }
    // insert/link n before c, if c exists
    if (c != null) {
```

Zusatzseite