Technische Universität Wien
Institut für Computergraphik und Algorithmen
Algorithms and Complexity Group

186.813 Algorithmen und Datenstrukturen 1 VU 6.0

2. Test SS 2017

1. Juni 2017

Machen Sie die folgenden Angaben bitte in deutlicher Blockschrift:

Nachname: | Vorname:

Matrikelnummer: | Unterschrift:

Legen Sie während der Prüfung Ihren Ausweis für Studierende vor sich auf das Pult.

Sie dürfen die Lösungen nur auf die Angabeblätter schreiben, die Sie von der Aufsicht erhalten. Es ist nicht zulässig, eventuell mitgebrachtes eigenes Papier zu verwenden. Benutzen Sie bitte dokumentenechte Schreibgeräte (keine Bleistifte!).

Die Verwendung von Taschenrechnern, Mobiltelefonen, Tablets, Digitalkameras, Skripten, Büchern, Mitschriften, Ausarbeitungen oder vergleichbaren Hilfsmitteln ist unzulässig.

	A1:	A2:	A3:	Summe:
Erreichbare Punkte:	16	18	16	50
Erreichte Punkte:				

Viel Erfolg!

Aufgabe A1: Hashtabellen **(16 Punkte)**

Fügen Sie die folgenden Zahlen in die jeweiligen Hashtabellen ein, indem Sie die angegebenen Hashfunktionen und Strategien für die Kollisionsbehandlung benutzen.

a) (4 Punkte)

Einzufügende Zahl: 16
Kollisionsbehandlung: Quadratisches Sondieren mit $c_1 = 1$ und $c_2 = 1$

Hashfunktion:
$h'(k) = k \bmod 7$

Hashtabelle:

0	1	2	3	4	5	6
		23		32	40	

b) (4 Punkte)

Einzufügende Zahl: 16
Kollisionsbehandlung: Double Hashing **ohne** die Verbesserung nach Brent

Hashfunktionen:
$h_1(k) = k \bmod 7$
$h_2(k) = (k \bmod 6) + 1$

Hashtabelle:

0	1	2	3	4	5	6
		23		32	40	

c) (4 Punkte)

Einzufügende Zahl: 79
Kollisionsbehandlung: Double Hashing **mit der Verbesserung nach Brent**
Wird ein bereits vorhandenes Element verschoben, so muss die neue Position dieses Elementes eindeutig gekennzeichnet werden.

Hashfunktionen:
$h_1(k) = k \bmod 7$
$h_2(k) = (k \bmod 6) + 1$

Hashtabelle:

0	1	2	3	4	5	6
		44		32	40	

d) (4 Punkte)

Gegeben sind im Folgenden mehrere Hashverfahren. Geben Sie zu jedem an, ob es gut funktionieren würde, oder ob es zu Problemen kommen könnte. Begründen Sie ihre Antworten.

I) (2 Punkte)

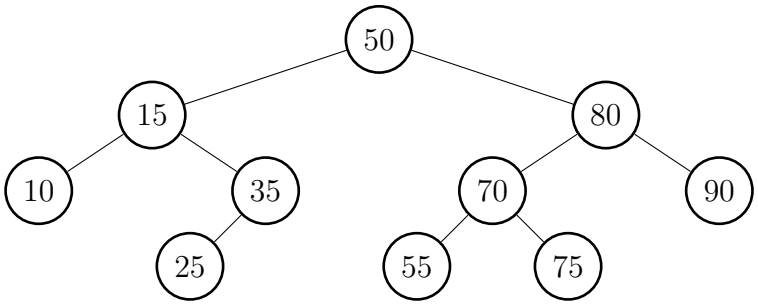
Multiplikationsmethode
Tabellengröße $m = 29$
Faktor $A = 3.25$

II) (2 Punkte)

Divisions-Rest-Methode mit Double Hashing
Tabellengröße $m = 40$
$h_1(k) = k \bmod 40$
$h_2(k) = (k \bmod 36) + 1$

Aufgabe A2: Bäume **(18 Punkte)**

a) (10 Punkte) Gegeben sei folgender AVL-Baum:



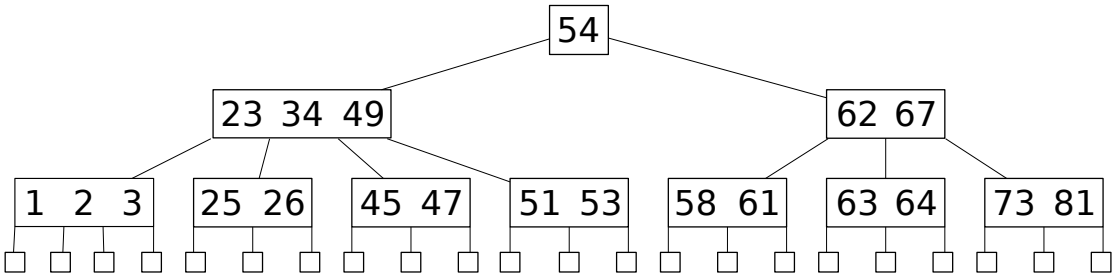
Teilen Sie alle noch nicht vorhandenen ganzen Zahlen aus dem Intervall $[0, 100]$ in Teilintervalle auf, sodass jede Zahl aus demselben Teilintervall an der gleichen Position in dem gegebenen Baum eingefügt werden würde. Tragen Sie nun diese Intervalle zusammen mit der entsprechenden Einfügestelle sowie die Art der Reorganisation, welche nach dem Einfügen einer Zahl in einem solchen Intervall ausgeführt werden muss, in die folgende Tabelle ein. Die erste Zeile in der Tabelle ist bereits exemplarisch ausgefüllt.

Intervall	Einfügestelle	Art der Reorganisation		
		keine	einfache Rotation	doppelte Rotation
0–9	links von 10	✓		

b) (8 Punkte) Gegeben sei der nachfolgende **B-Baum der Ordnung 5**.

- Entfernen Sie aus dem angegebenen Baum den Schlüssel 26.
- Entfernen Sie aus dem angegebenen Baum den Schlüssel 62.

Geben Sie dabei alle Zwischenschritte an! Sie können sich bei der Angabe der Zwischenschritte auf den Teilbaum beschränken, der sich geändert hat.

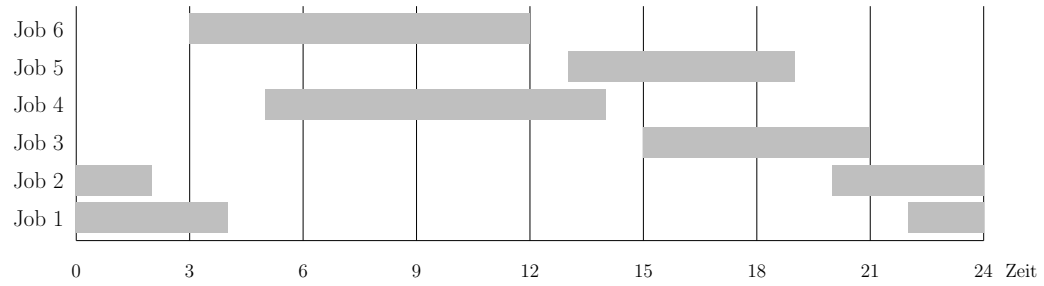


Aufgabe A3: Greedy **(16 Punkte)**

Betrachten Sie die folgende Modifikation des Interval-Scheduling-Problems. Ein Prozessor operiert 24 Stunden am Tag, jeden Tag. Leute stellen Anfragen, um tägliche Jobs (Programme) auf dem Prozessor auszuführen. Jeder solcher Jobs hat eine Startzeit und eine Endzeit (welche z.B. in Sekunden nach Mitternacht angegeben werden kann). Wird der Job zum Ausführen akzeptiert, dann muss er täglich von der angegeben Startzeit bis zur angegebenen Endzeit ausgeführt werden. Beachten Sie hier, dass ein Job auch vor Mitternacht beginnen und erst am nächsten Tag enden kann. Weiterhin darf jeder Job höchstens 24 Stunden dauern.

Entwerfen Sie einen Algorithmus mit Laufzeit $O(n^2 \log n)$, welcher ähnlich zu dem aus der Vorlesung bekannten Greedy-Algorithmus für das normale Interval-Scheduling-Problem, eine größenmässig maximale Teilmenge von Jobs findet, welche kontinuierlich (über mehrere Tage hinweg) ohne Überschneidungen ausgeführt werden kann.

Die folgende Abbildung zeigt eine mögliche Instanz dieses Problems mit 6 Jobs. Die Beispielinstanz hat zwei optimale Lösungen: die Teilmenge die aus den Jobs 1, 3 und 4 besteht sowie die Teilmenge die aus den Jobs 2, 5 und 6 besteht.



Wir sagen ein Job ist "einfach", falls er vollständig innerhalb eines Tages ausgeführt werden kann. Im obigen Beispiel sind die Jobs 3–6 einfache Jobs, die Jobs 1 und 2 hingegen nicht.

a) (1 Punkt) Geben Sie ein einfaches Kriterium an, welches es Ihnen erlaubt zu entscheiden, ob ein Job einfach ist:

b) (2 Punkte) Wir sagen eine Instanz des modifizierten Interval-Scheduling-Problems ist einfach, falls sie nur aus einfachen Jobs besteht. Stimmt es, dass der Ihnen aus der Vorlesung bekannte Greedy-Algorithmus für das Interval-Scheduling-Problem, die richtige Lösung für jede einfache Instanz liefert? Begründen Sie Ihre Antwort!

c) (1 Punkt) Wieviele nicht einfache Jobs kann eine optimale Lösung des modifizierten Interval-Scheduling-Problems maximal enthalten?

d) (12 Punkte) Schreiben Sie nun in detailiertem Pseudocode eine Funktion

`Menge findeMaximaleMenge( Menge einfJobs, Menge schwJobs )`,

welche eine optimale Lösung für das modifizierte Interval-Scheduling-Problem zurückgibt und eine Laufzeit von höchstens $O(n^2 \log n)$ für eine gegebene Menge von n Jobs aufweist. Die beiden Parameter `einfJobs` und `schwJobs` der Funktion enthalten dabei die Menge aller einfachen bzw. die Menge aller nicht einfachen Jobs der Instanz.

Sie können dafür die Ihnen aus der Vorlesung bekannte Funktion, nachfolgend als `Greedy` bezeichnet, in der Form `Menge Greedy(Menge J)` als gegeben voraussetzen, welche für eine gegebene Menge J von Jobs, eine optimale Teilmenge für das (unmodifizierte) Interval-Scheduling-Problem zurückgibt und eine Laufzeit von $O(|J| \log |J|)$ aufweist.

Die Datenstruktur `Menge` einer Menge m unterstützt die folgenden Operationen:

- `m.size()` gibt die Größe der Menge m zurück.
- `m.add( Job j )` fügt den Job j zur Menge m hinzu.

Desweiteren können Sie alle Elemente der Menge m mit der Anweisung `foreach j` $\in$ `m` durchlaufen. Auf die Startzeit und Endzeit eines Jobs j können Sie mit `j.start` und `j.end` zugreifen.