Nachname:	Vorname:	Matrikelnummer:

↑ Bitte **sehr deutlich** schreiben und BLOCKBUCHSTABEN (groß geschriebene Druckbuchstaben) verwenden ↑

Prof. für Algorithmen
und Datenstrukturen
Prof. Dr. Hannah Bast
Dr. Patrick Brosi

Algorithmen und Datenstrukturen
SS 2023

http://ad-wiki.informatik.uni-freiburg.de/teaching

UNI FREIBURG

Klausur

Dienstag 8. August 2023, 14:00 - 16:30 Uhr, in den Räumen von Gebäude 101 der Technischen Fakultät

Es gibt *fünf* Aufgaben. Für jede davon gibt es maximal 20 Punkte. Sie können also maximal 100 Punkte erreichen. Zum Bestehen reichen 50 Punkte. Sie haben insgesamt zweieinhalb Stunden Zeit. Damit haben Sie im Durchschnitt 30 Minuten Zeit pro Aufgabe. Jede Aufgabe sollte in 20 Minuten machbar sein, die restlichen 10 Minuten sind als Puffer gedacht.

An Lehrmaterialien dürfen Sie maximal **ein** DIN-A4 Blatt Papier verwenden, das beidseitig mit von Ihnen selber zusammengestellten Inhalten beschriftet oder bedruckt sein kann. Sie dürfen keinerlei elektronische Geräte verwenden, insbesondere keine Geräte, mit denen Sie mit Dritten kommunizieren oder sich mit dem Internet verbinden können.

Wir werden die Klausur am 9. und 10. August korrigieren und die Noten direkt im Anschluss im HISinOne eintragen. Die Klausureinsicht wird am Freitag, den 11. August stattfinden. Genauere Informationen dazu werden noch auf dem Forum bekannt gegeben.

Bemerkung zu den Programmieraufgaben: Wenn Code gefragt ist, können Sie frei zwischen Python, Java und C++ wählen. Wir empfehlen Python. Für kleinere, rein syntaktische Fehler gibt es keinen Punktabzug. Alle Programmieraufgaben dieser Klausur lassen sich mit wenig Code lösen. Pro Aufgabe ist eine Obergrenze für die Anzahl der Zeilen vorgegeben.

Bemerkung zu den Multiple-Choice-Aufgaben: Bei einigen der Aufgaben wird lediglich nach dem Wahrheitswert von Aussagen gefragt. Antworten Sie dann zu jeder Aussage mit den Worten *wahr* oder *falsch* oder nichts. Ohne Antwort gibt es für die betreffende Aussage 0 Punkte. Für eine richtige Antwort gibt es 2 Punkte, für eine falsche Antwort gibt es zwei Punkte Abzug. Insgesamt können Sie für eine solche Aufgabe nicht weniger als 0 Punkte bekommen. Achten Sie darauf, dass in Ihrer Abgabe klar ist, zu welcher Aussage eine Antwort gehört.

Was Sie wie abgeben sollen: Schreiben Sie bitte oben in die blaue Box Ihren Namen und Ihre Matrikelnummer. Schreiben Sie Ihre Lösungen bitte auf die Klausur: benutzen Sie dabei für jede Aufgabe zuerst die Vorderseite des Blattes, auf dem die Aufgabe steht und dann die Rückseite des *vorherigen* Blattes. Wenn Sie zusätzliches Papier abgeben wollen (das sollte die Ausnahme sein), schreiben Sie auf jedes zusätzliche Blatt Ihren Namen und Ihre Matrikelnummer.

Wir wünschen Ihnen viel Erfolg!

A1	A2	A3	A4	A5	Punkte	Note

Aufgabe 1 (Sortieren, 20 Punkte)

1.1 (6 Punkte) Schreiben Sie eine Funktion *sort_one(lst, i)*, die eine Liste *lst*, die mit Ausnahme des Elements an Position i bereits aufsteigend sortiert ist, vollständig aufsteigend sortiert. Sei n die Länge von *lst*. Sie können annehmen, dass $n \geq 2$ und $0 \leq i < n$. Die Funktion soll in Zeit $O(n)$ laufen und maximal 10 Zeilen lang sein.

```
def sort_one(lst: list, i: int):
   while i > 0 and lst[i] < lst[i-1]:
       lst[i], lst[i-1] = lst[i-1], lst[i]
       i -= 1
   while i < len(lst) - 1 and lst[i] > lst[i+1]:
       lst[i], lst[i+1] = lst[i+1], lst[i]
       i += 1
```

1.2 (6 Punkte) Wir betrachten Eingaben, die aus Key-Value Paaren bestehen, deren Keys alle gleich sind. Ein Sortieralgorithmus wird für solche Eingaben *stabil* genannt, wenn er die Elemente in der gleichen Reihenfolge zurückgibt, in der sie in der Eingabe stehen. Beweisen oder widerlegen Sie: *HeapSort* (mittels eines binären Heaps) ist für solche Eingaben stabil.

1. Die Eingabe erfüllt trivialerweise die Heap-Eigenschaft.
2. Durch das inititale *heapify* (oder n-mal *insert*) ändert sich also nichts.
3. Der erste Aufruf von *get_min* gibt das Element an der Wurzel des Heaps aus.
4. Der folgende Aufruf von *delete_min* setzt dann das letzte Element der Eingabe an die Wurzel.
5. Der folgende zweite Aufruf von *get_min* gibt also das letzte Element der Eingabe aus.
6. Für mehr als drei Elemente ist damit die Reihenfolge der Elemente verändert worden.
7. Damit ist *HeapSort* nicht stabil

1.3 (8 Punkte) Geben Sie für jede der folgenden Aussagen an, ob sie *wahr* oder *falsch* ist. Eine richtige Antwort gibt 2 Punkte, eine falsche Antwort gibt 2 Punkte Abzug. Wenn sie die Antwort nicht wissen, schreiben Sie *keine Aussage* oder nichts. Sie können nicht weniger als 0 Punkte für diese Aufgabe bekommen.

1. Die Laufzeit von *MergeSort* ist für alle Eingaben der Größe n in $\Theta(n \cdot \log n)$.
2. Die Eingabegröße für *MergeSort* muss eine Zweierpotenz sein.
3. Die Anzahl der Vergleiche von *MinSort* ist abhängig von der Reihenfolge der Eingabeelemente.
4. *BogoSort* ist für jede Eingabe langsamer als *HeapSort*.

1. Wahr $\rightarrow$ für jede Eingabe der Größe n gibt es $\Theta(\log n)$ Runden mit Laufzeit je $\Theta(n)$.
2. Falsch $\rightarrow$ für Zweierpotenzen ist es leichter zu erklären, aber es funktioniert für alle n.
3. Falsch $\rightarrow$ für jede Eingabe wird in Runde i gegen jedes Element ab Position i verglichen.
4. Falsch $\rightarrow$ die Laufzeit von *HeapSort* ist immer $\Theta(n \log n)$, die von *BogoSort* kann $O(n)$ sein.

Aufgabe 2 (I/E-Sequenzen und O-Notation, 20 Punkte)

2.1 (6 Punkte) Die folgende Funktion sortiert die gegebene Liste absteigend. Geben Sie die `I`/`E`-Sequenz für die Eingabe $[1, 5, 3]$ an und schreiben Sie dabei *über* jedes `I` bzw. `E` die zugehörige Zeilennummer des Programms. Die *for*-Schleifen sollen Sie dabei *nicht* umschreiben, sondern wie folgt berücksichtigen: wenn eine Iteration einer *for*-Schleife ausgeführt wird, entspricht dies einem `E`, sonst einem `I`, jeweils mit der Zeilennummer der *for*-Schleife darüber.

```
1. def bubble_sort(nums: list[int]):
2.   upper = len(nums) - 1
3.   for i in range(upper):
4.     for k in range(i, upper):
5.       if nums[k] < nums[k + 1]:
6.         nums[k+1], nums[k] = nums[k], nums[k+1]
```

1. Die Sequenz der *for*-Schleife in Zeile 3 ist (unabhängig von der Eingabe) `EEI`.
2. Die Sequenz der *for*-Schleife in Zeile 4 ist (unabhängig von der Eingabe) `EEIEI`.
3. Die Sequenz von Zeile 5 beim ersten Durchlauf der inneren Schleife ist `II` (danach: $[5, 3, 1]$).
4. Die Sequenz von Zeile 5 beim zweiten Durchlauf der inneren Schleife ist `E` (danach: unverändert).
5. Die Gesamtsequenz ist damit:

3	4	5	4	5	4	3	4	5	4	3
E	E	I	E	I	I	E	E	E	I	I

2.2 (6 Punkte) Geben Sie für jede der folgenden Aussagen an, ob sie *wahr* oder *falsch* ist. Eine richtige Antwort gibt 2 Punkte, eine falsche Antwort gibt 2 Punkte Abzug. Wenn Sie die Antwort nicht wissen, schreiben Sie *keine Aussage* oder nichts. Sie können insgesamt nicht weniger als 0 Punkte für diese Aufgabe bekommen.

1. $n^3 + 6n^2 + n \cdot \log n \in \Theta(0.5n^3)$ — Wahr $\to \lim_{n\to\infty}(n^3 + 6n^2 + n)/(0.5n^3) = 0.5$
2. $n! \in O(n^n)$ — Wahr $\to \lim_{n\to\infty} n!/n^n = 0$
3. $\log(n!) \in o(n \log n)$ — Falsch $\to \lim_{n\to\infty} \log(n!)/(n \log n) = 1$

2.3 (8 Punkte) Seien f_1, f_2, f_3 Funktionen mit $f_1 \in \mathrm{O}(f_2)$ und $f_2 \in \mathrm{o}(f_3)$. Beweisen Sie, dass $f_1 \in o(f_3)$. Argumentieren Sie über die Definitionen von O und o und **nicht** über Grenzwerte.

1. $f_1 \in o(f_3)$ heißt: für alle $C > 0$ gibt es $n_0 > 0$, sodass $f_1(n) \leq C \cdot f_3(n)$ für alle $n > n_0$.
2. Sei also $C > 0$ beliebig.
3. $f_1 \in O(f_2) \to$ es gibt $n_1, C_1 > 0$, sodass $f_1(n) \leq C_1 \cdot f_2(n)$ für alle $n > n_1$.
4. $f_2 \in o(f_3) \to$ für $C_2 = C/C_1 > 0$ gibt es $n_2 > 0$, sodass $f_2(n) \leq C_2 \cdot f_2(n)$ für alle $n > n_2$.
5. Für $n_0 = \max\{n_1, n_2\}$ gilt, dass $f_1(n) \leq C_1 \cdot f_2(n) \leq C_1 \cdot C_2 \cdot f_3(n) = C \cdot f_3(n)$ für alle $n > n_0$.

Aufgabe 3 (Assoziative und dynamische Felder, 20 Punkte)

3.1 (5 Punkte) Sei $h(x) = (26 \cdot x - 137) \bmod 5$ eine Hashfunktion. Berechnen Sie den Hashwert $h(x)$ für jeden Schlüssel x der Folge $1907, 1934, 1953, 1981, 1999$. Die Schlüssel werden in dieser Reihenfolge in eine Hashtabelle der Größe 5 eingefügt, mittels der Hashfunktion h und offener Adressierung. Zeichnen Sie den Zustand der Hashtabelle nach dem letzten Einfügen.

1. $h(x) = x - 2 \bmod 5$, weil $26 \bmod 5 = 1$ und $137 \bmod 5 = 2$.
2. $h(1907) = h(2) = 0$, $h(1934) = h(1999) = h(4) = 2$, $h(1953) = h(3) = 1$, $h(1981) = h(1) = 4$.
3. Zustand der Hashtabelle nach Einfügen aller Schlüssel: $[1907, 1953, 1934, 1999, 1981]$.

3.2 (7 Punkte) Für ein dynamisches Feld sei s_i die Anzahl der Elemente und c_i die Kapazität, jeweils nach Operation O_i. Sei die Vergrößerungsstrategie so, dass wenn vor einem *push* $c_{i-1} = s_{i-1}$, dann wird die Kapazität verdoppelt ($c_i = 2 \cdot c_{i-1}$). Sei die Verkleinerungsstrategie so, dass wenn vor einem *pop* $s_{i-1} \leq c_{i-1}/2$, dann wird die Kapazität halbiert ($c_i = \lceil c_{i-1}/2 \rceil$). Geben Sie für $n \in \mathbb{N}$ eine Folge von $4n$ Operationen an, deren Kosten $\Omega(n^2)$ sind, wenn zu Beginn $s_0 = c_0 = n$ (das n darf dabei nicht als konstant angenommen werden). Geben Sie die Werte von s_i und c_i nach jeder Operation an.

1. $O_1 = push$ $\rightarrow$ es wird verdoppelt ($s_0 = c_0$) $\rightarrow s_1 = n + 1$, $c_1 = 2n$.
2. $O_2 = pop$ $\rightarrow$ es wird nicht halbiert ($s_1 > c_1/2 = n$) $\rightarrow s_2 = n$ und $c_2 = 2n$.
3. $O_3 = pop$ $\rightarrow$ es wird halbiert ($s_2 = c_2/2 = n$) $\rightarrow s_3 = n - 1$ und $c_3 = n$.
4. $O_4 = push$ $\rightarrow$ es wird nicht verdoppelt ($s_3 < c_3$) $\rightarrow s_4 = n$ und $c_4 = n$.
5. Jetzt sind wir in derselben Situation wie am Anfang und können das n mal wiederholen.
6. Jeder Viererzyklus hat Kosten $\geq n$, alle $4n$ Operationen kosten also $\Omega(n^2)$.

3.3 (8 Punkte) Wir hatten in der Vorlesung ein Mastertheorem formuliert für eine Folge von n Operationen $O_1, \ldots, O_n$, Laufzeit T_i für Operation O_i, Potenzial Φ_i nach O_i, und Potential Φ_0 am Anfang. Die Bedingungen waren, dass $\Phi_i \geq 0$ und $T_i \leq A + B \cdot (\Phi_{i-1} - \Phi_i)$ für $i = 1, \ldots, n$ und Konstanten $A, B > 0$.

Geben Sie für jede der folgenden Aussagen an, ob sie *wahr* oder *falsch* ist. Eine richtige Antwort gibt 2 Punkte, eine falsche Antwort gibt 2 Punkte Abzug. Wenn sie die Antwort nicht wissen, schreiben Sie *keine Aussage* oder nichts. Sie können nicht weniger als 0 Punkte für diese Aufgabe bekommen.

1. Wenn die Bedingungen erfüllt sind, dann gilt $\sum_{i=1}^{n} T_i = O(n + \Phi_0)$.
2. Wenn die Bedingungen erfüllt sind, aber statt $\Phi_i \geq 0$ ist $\Phi_n \geq \Phi_0$, dann gilt $\sum_{i=1}^{n} T_i = O(n)$.
3. Wenn die Φ_i alle negativ sind, kann die Bedingung $T_i \leq A + B \cdot (\Phi_{i-1} - \Phi_i)$ nicht erfüllt sein.
4. Wenn alle $T_i = O(1)$ für alle i, dann erfüllt $\Phi_i = i$ die Bedingungen für geeignete $A, B > 0$.

1. Wahr $\rightarrow$ das ist genau die Aussage, die wir in der Vorlesung bewiesen haben.
2. Wahr $\rightarrow$ der Beweis über die Teleskopsumme zeigt, dass $\sum_{i=1}^{n} T_i = O(n + \Phi_0 - \Phi_n)$.
3. Falsch $\rightarrow$ die Bedingung kann problemlos auch für negative Φ_i erfüllt sein.
4. Wahr $\rightarrow$ für $\Phi_i = i$ ist die Bedingung an T_i, dass $T_i \leq A - B$ und A und B sind frei wählbar.

Aufgabe 4 (Blockoperationen und binäre Suchbäume, 20 Punkte)

4.1 (6 Punkte) Sei A ein Feld mit n Elementen, das an einer Blockgrenze anfängt. Über die n Elemente des Feldes wird in zufälliger Reihenfolge iteriert. Sei M die Größe des schnellen Speichers und B die Blockgröße. Wie viele Blockoperationen werden im besten Fall benötigt und wie viele im schlechtesten Fall, wenn $M = 2 \cdot B$ und $n \gg M$? Mit Begründung!

1. Bester Fall: $\lceil n/B \rceil$. Wenn z.B. in geordneter Reihenfolge über die Elemente des Feldes iteriert wird. Besser geht es nicht, denn jeder Block des Feldes muss mindestens einmal geladen werden.
2. Schlechtester Fall: n. Wenn aus jedem Block jeweils nur ein Element gelesen wird bevor ein Element aus dem nächsten Block gelesen wird. Wegen $n \gg M$ ist dies möglich, ohne dass jemals noch etwas Brauchbares im Cache steht. Schlechter geht es nicht, denn es müssen nie mehr Blöcke geladen werden, als es Elemente im Feld gibt.

4.2 (6 Punkte) Ein binärer Suchbaum mit Tiefe d heißt *fast vollständig*, wenn er auf jeder Ebene die maximal mögliche Anzahl an Knoten hat, außer auf der tiefsten Ebene (ein Baum, der nur aus der Wurzel besteht, hat Tiefe 0). Geben Sie die minimale und maximale Anzahl an Knoten eines fast vollständiger Baumes der Tiefe d in Abhängigkeit von d an. Mit Begründung!

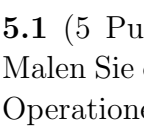
1. Für Tiefe $i < d$ haben wir genau 2^i Knoten (die maximale Anzahl für Tiefe i).
2. Die Gesamtzahl Knoten in den Ebenen 0 bis $d-1$ ist also $\sum_{i=0}^{d-1} 2^i = 2^d - 1$.
3. Die letzte (nicht vollständige) Ebene hat mindestens einen Knoten und höchstens $2^d - 1$ Knoten.
4. Daher $n_{min} = 2^d$ und $n_{max} = 2^{d+1} - 2$

4.3 (8 Punkte) Schreiben Sie eine rekursive Funktion *is_valid_bst(node: BinarySearchTreeNode)*, die den Wurzelknoten eines binären Baumes als Argument nimmt und prüft, ob der zugehörige Baum ein valider binärer Suchbaum ist. Sie können annehmen, dass ein *BinarySearchTreeNode* die folgenden Attribute hat: *key* (der Schlüssel, ein Integer), *value* (der Wert, ein beliebiges Objekt), *left* (linkes Kind, ein *BinarySearchTreeNode* oder *None*) und *right* (rechtes Kind, ein *BinarySearchTreeNode* oder *None*). Die Funktion soll maximal 10 Zeilen lang sein.

```
def is_valid_bst(node, min_node=None, max_node=None):
    if node is None:
        return True
    if min_node and node.key < min_node.key:
        return False
    if max_node and node.key > max_node.key:
        return False
    return is_valid_bst(node.left, min_node, node) and \
           is_valid_bst(node.right, node, max_node)
```

Aufgabe 5 (Heaps und Graphen, 20 Punkte)

5.1 (5 Punkte) Sei $A = [\times, 3, 5, 8, 12, 7]$ die Speicherung eines binären Heaps in einem Feld. Malen Sie den initialen Zustand des Heaps als Baum. Führen Sie dann nacheinander die folgenden Operationen auf dem Heap aus und geben Sie den Zustand des Feldes A nach *jeder* Operation an: *insert(6)*, *delete_min()*, *delete_min()*, *insert(1)*.



insert(6): $[\times, 3, 5, 6, 12, 7, 8]$
delete_min(): $[\times, 5, 7, 6, 12, 8]$
delete_min(): $[\times, 6, 7, 8, 12]$
insert(1): $[\times, 1, 6, 8, 12, 7]$

5.2 (10 Punkte) Schreiben Sie eine Funktion *merge(lists)*, die eine Liste von k jeweils aufsteigend sortierten, nicht-leeren Listen von Zahlen als Eingabe akzeptiert und eine einzige aufsteigend sortierte Liste zurückgibt, die genau die Elemente aus allen Listen enthält. Ihre Funktion soll maximal 15 Zeilen umfassen und in Zeit $O(n \cdot \log k)$ laufen, wobei n die Gesamtzahl der Elemente ist. Sie dürfen eine Klasse *PriorityQueue* annehmen, mit einer Operation *get_min*, deren Laufzeit konstant ist, und Operationen *insert* und *delete_min*, deren Laufzeit jeweils $O(\log m)$ ist, wenn m Elemente in der PriorityQueue gespeichert sind.

```
def merge(lists):
  res = []
  pq = PriorityQueue()
  for i in range(0, len(lists)):
    pq.insert((lists[i][0], (i, 0)))
  while pq.get_min() != None:
    min = pq.get_min()
    pq.delete_min()
    res.append(min[0])
    if min[1][1] + 1 < len(lists[min[1][0]]):
      pq.insert((lists[min[1][0]][min[1][1]+1], (min[1][0], min[1][1] + 1)))
  return res
```

5.3 (5 Punkte) Sei $P = u_1, \ldots, u_k$ ein kürzester Pfad von u_1 nach u_k durch einen Graph $G = (V, E)$, wobei $k \geq 2$ und $u_1, \ldots, u_k \in V$. Beweisen oder widerlegen Sie: der Pfad $Q = u_1, \ldots, u_{k-1}$ ist ebenfalls ein kürzester Pfad von u_1 nach u_{k-1}.

1. Angenommen, es gäbe einen kürzeren Pfad Q' von u_1 nach u_{k-1}.
2. Sei P' der Pfad, der aus P hervorgeht, wenn man den Teilpfad Q durch Q' ersetzt.
3. Wenn Q' kürzer ist als Q, dann ist auch P' kürzer als P.
4. Das widerspricht der Annahme, dass P ein kürzester Pfad war.
5. Es kann also kein solches Q' geben, also ist Q auch ein kürzester Pfad.