Aufgabe 1: Mehrfachauswahlfragen (/10 Punkte)

Gegeben ist ein Eingabearray A mit n Elementen. Der Wert m beschreibt die Größe des Wertebereichs der möglichen Elemente in A. Kreuzen Sie jeweils die richtige Antwort an.

Grundlagen der Sortieralgorithmen

(a) (1 Punkt) Welcher Ausdruck beschreibt am genauesten, wie viele Vergleiche *SelectionSort* im Worst-Case benötigt?

○ $O(n)$
○ $O(n \log n)$
⊗ $O(n^2)$
○ $O(\log n)$

(b) (1 Punkt) Welche Aussage beschreibt korrekt, wann *InsertionSort* zum aufsteigend Sortieren in $\Theta(n)$ Zeit läuft?

○ Wenn das Array streng absteigend sortiert ist.
⊗ Wenn das Array bereits fast sortiert ist.
○ Wenn das Array keine doppelten Werte enthält.
○ Wenn die Hälfte der Werte im Array doppelt vorkommen.

(c) (1 Punkt) Welches Verfahren garantiert beim Sortieren von Ganzzahlen im Bereich $0..m$ für $m \leq n$ Laufzeit linear in n?

○ QuickSort
○ MergeSort
⊗ CountSort
○ HeapSort

Sortieralgorithmus-Interna und Pseudocode

(a) (1 Punkt) Bei *BubbleSort* wird in jedem Durchlauf das größte Element in den rechten Bereich verschoben. Was ist eine korrekte Invariante nach Abschluss der i-ten äußeren Schleifeniteration?

○ $A[1..i]$ ist sortiert.
⊗ Die letzten i Elemente sind die größten und in richtiger Reihenfolge.
○ Die ersten i Elemente sind die kleinsten Elemente in richtiger Reihenfolge.
○ Das gesamte Array ist sortiert.

(b) (1 Punkt) Ein Pseudocode-Ausschnitt lautet:

```
for i = 1 to n do push(S, A[i])
```

Wenn S ein Stack ist, welcher der folgenden Ausdrücke beschreibt den Inhalt von S nach Ausführung der Schleife (ältestes Element links, neuestes Element rechts)?

⊗ $[A[1], A[2], \ldots, A[n]]$
○ $[A[n], A[n-1], \ldots, A[1]]$
○ $[A[1]]$
○ Ein undefinierter Zustand. Es hängt vom Compiler ab.

(c) (1 Punkt) Betrachte folgendes Pseudocode-Snippet:

```
for i = 1 to n do enqueue(Q, A[i])
```

Angenommen, Q ist eine FIFO-Queue. Nachdem der Pseudocode vollständig ausgeführt wurde, welcher Ausdruck beschreibt die Ausgabe, wenn man `dequeue(Q)` wiederholt nacheinander aufruft, bis `Q` leer ist?

⊗ $A[1], A[2], \ldots, A[n]$
○ $A[n], A[n-1], \ldots, A[1]$
○ Zufällige Reihenfolge
○ Nur $A[1]$, danach Fehlerzustand

Einfache Datenstrukturen

(a) (1 Punkt) Was ist die Worst-Case-Laufzeit, um in einer *einfach verketteten Liste* von n sortierten, paarweise verschiedenen Zahlen einen bestimmten Wert zu finden? Wir verfügen über einen Zeiger auf das erste Element.

○ $O(1)$
○ $O(\log n)$
⊗ $O(n)$
○ $O(n \log n)$

(b) (1 Punkt) Welche Operation ist bei einem (binären) *Suchbaum* ohne Balancierung im Worst-Case am aufwendigsten in asymptotischer Laufzeit?

⊗ Einfügen eines neuen Wertes
○ Zugriff auf das linke Kind eines bekannten Knotens (Zeiger vorhanden)
○ Zugriff auf die Wurzel
○ Austauschen der Werte zweier bekannter Blattknoten (Zeiger vorhanden)

Fortgeschrittene Datenstrukturen

(a) (1 Punkt) Was ist die asymptotische Worst-Case-Laufzeit, um in einem AVL-Baum von n Zahlen festzustellen, ob ein bestimmter Wert vorhanden ist?

○ $O(1)$
⊗ $O(\log n)$
○ $O(n)$
○ $O(n \log n)$

(b) (1 Punkt) Was ist die asymptotische Worst-Case-Laufzeit von k-maligem *Heap-Extract-Max* auf einem Max-Heap mit n Elementen?

○ $O(k)$
○ $O(k \log k)$
⊗ $O(k \log n)$
○ $O(n \log n)$

Aufgabe 2: Datenstrukturen (/4 Punkte)

Wir betrachten eine **Queue**, die nach dem FIFO-Prinzip arbeitet und als **Ringpuffer** (Circular Buffer) auf einem 1-indizierten Array implementiert ist. Die Queue verwaltet zwei Zustandsvariablen:

- `head`: Der Index des ältesten Elements (Bereich 1 bis n).
- `count`: Die aktuelle Anzahl der Elemente in der Queue.

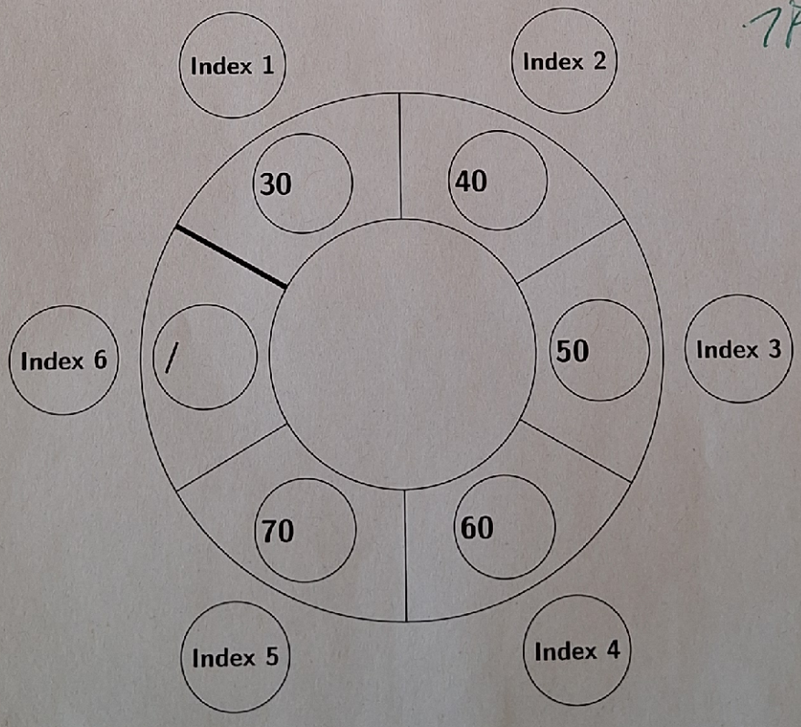
Die Ringpuffer-Operationen funktionieren wie folgt:

- **`enqueue(x)`**: Sofern `count < n`, fügt `x` am Index `((head - 1 + count) modulo n) + 1` ein und erhöht `count` um 1.
- **`dequeue()`**: Sofern `count > 0`, entfernt das Element am Index `head`, setzt `head` auf `(head modulo n) + 1` und verringert `count` um 1.

(a) (3 Punkte) Gegeben ist ein Ringpuffer der Größe `n=6` mit den Werten `head=5` und `count=3`. Die Queue enthält die Elemente `[10, 20, 30]`, wobei 10 das älteste Element und 30 das neueste Element ist. Führen Sie die folgenden Operationen nacheinander aus.

a.) `enqueue(40)`
b.) `dequeue()`
c.) `enqueue(50)`
d.) `enqueue(60)`
e.) `dequeue()`
f.) `enqueue(70)`

Tragen Sie die Werte des **Endzustands** des Ringpuffers ein. Streichen Sie freie Felder durch.



1P - korrekte Werte (Reihenfolge egal)
1P - Aufsteigend
u. zusammenhängend
1P - Block start = Block ende = 6 leer

Lösungsweg:

- Start: Array[_, _, _, _, 10, 20], Array[1]=30. Head=5, Count=3.
- `enq(40)`: Index $((5-1+3)\%6)+1=2$. Array[2]=40. Count=4.
- `deq()`: Entfernt Array[5]=10. Head=$(5\%6)+1=6$. Count=3.
- `enq(50)`: Index $((6-1+3)\%6)+1=3$. Array[3]=50. Count=4.
- `enq(60)`: Index $((6-1+4)\%6)+1=4$. Array[4]=60. Count=5.

ML.
head=7, count=3
0,5 P 0,5 P

- `deq()`: Entfernt Array[6]=20. Head=(6%6) + 1 = 1. Count=4.
- `enq(70)`: Index ((1 − 1 + 4)%6) + 1 = 5. Array[5]=70. Count=5.
- **Endzustand:** Werte [30, 40, 50, 60, 70] an Indizes 1, 2, 3, 4, 5. Index 6 ist leer.

Bewertung:

- 1P für die korrekten Werte: Die Zahlen 30, 40, 50, 60, 70 sind final im Ring (egal wo).
- 1P für die korrekte Reihenfolge: Die Zahlen bilden einen zusammenhängenden Block in aufsteigender Folge im Uhrzeigersinn.
- 1P für die korrekte Position: Der Block beginnt bei Index 1 und Index 6 ist leer.

(b) (1 Punkt) Was sind die finalen Werte für `head` und `count` nach diesen Operationen?

`head:` 1 `count:` 5

Bewertung:

- Je 0.5 Punkte für korrekten Head und Count.
- Folgefehler beachten.

Aufgabe 3: Komplexitätsanalyse (/6 Punkte)

Gegeben: Ein Array A der Länge n. Betrachten Sie folgenden Pseudocode:

```
1    n ← len(A)
2    for i ← 1 to n-1 do
3     for j ← 1 to n-1-i do
4         if A[j] > A[j+1] then
5             swap(A[j], A[j+1])
```

(a) (1 Punkt) Welcher Sortieralgorithmus wird hier dargestellt?
Bubble Sort

(b) (4 Punkte) Geben Sie für jede Zeile des gegebenen Pseudocodes die Anzahl der Ausführungen im Best-Case in Abhängigkeit von n an. **Hinweis: gesucht sind konkrete Terme, keine asymptotische Laufzeiten. Also beispielsweise $7n + 5$, nicht $O(n)$**

Zeile	**Best Case**
1	1
2	n
3	$\frac{n \cdot (n-1)}{2}$
4	$\frac{(n-2) \cdot (n-1)}{2}$
5	0

Bewertung:

- Je Zeile 1 Punkt

(c) (1 Punkt) Welche der folgenden Aussagen zur Komplexität sind korrekt? (Zwei Antworten sind richtig.)

⊗ Die Gesamtzahl der Vergleiche (Zeile 4) ist im Worst-Case $O(n^2)$.
○ Der gegebene Algorithmus benötigt im Best-Case $O(n^2)$ Vertauschungen.
⊗ Wird Zeile 2 r-mal ausgeführt, so trägt sie $O(r)$ zur Gesamtlaufzeit bei.
○ Zeile 4 wird im Best Case $O(n)$-mal ausgeführt.

Aufgabe 4: HybridSort (/7 Punkte)

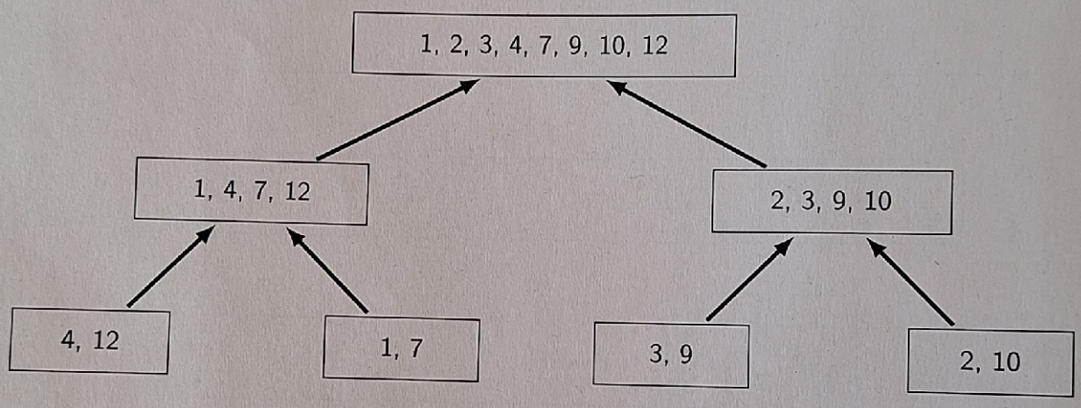
In dieser Aufgabe betrachten wir eine Variation von `MergeSort`, die wir `HybridSort` nennen. Die Funktion erhält das Array A, die Indizes `links` und `rechts` (Ganzzahlen) sowie den Schwellenwert k. Der Algorithmus wechselt zu `InsertionSort`, sobald der Bereich von `links` bis `rechts` höchstens k Elemente umfasst.

```
1  HybridSort(A, links, rechts, k)
2      if (1 + rechts − links) > k then
3          mitte ← (links + rechts) / 2    // Abrunden
4          HybridSort(A, links, mitte, k)
5          HybridSort(A, mitte + 1, rechts, k)
6          Merge(A, links, mitte, rechts)
7      else
8          InsertionSort(A, links, rechts)
```

(a) (4 Punkte) **Handsimulation**.

Gegeben sei das 1-indizierte Array $A = [12, 4, 7, 1, 9, 3, 10, 2]$. Der Schwellenwert ist $k = 3$.

Führen Sie den Algorithmus für den initialen Aufruf `HybridSort(A, 1, 8, 3)` aus. Tragen Sie von unten nach oben in den untenstehenden Rekursionsbaum den Inhalt des Teilarrays nach der Bearbeitung für jeden Knoten ein.



Bewertung:

- 0.5P pro korrekt sortiertem Blatt (`InsertionSort` Phase) = 2P.
- 0.5P pro korrektem Merge auf mittlerer Ebene = 1P.
- 1P für korrektes Endergebnis (Wurzel).

(b) (1 Punkt) **Komplexität**. Geben Sie die asymptotische Worst-Case-Laufzeit von `HybridSort` in der $\mathcal{O}$-Notation an, unter der Annahme, dass k eine Konstante ist.

$$\mathcal{O}\Big(\quad n \log n \quad \Big)$$

Bewertung: 1P für $n \log n$.

(c) (2 Punkte) **Analyse**. Warum ist diese hybride Variante in der Praxis oft schneller als das reine `MergeSort`? Kreuzen Sie die **zwei** korrekten Aussagen an.

◯	`InsertionSort` hat eine bessere Worst-Case-Laufzeit als `MergeSort` ($O(n)$ vs. $O(n \log n)$).
⊗	Für kleine n ist `InsertionSort` schneller, da es keinen Rekursions-Overhead hat.
◯	Der Speicherverbrauch von `HybridSort` ist $O(1)$, während `MergeSort` $O(n)$ benötigt.
⊗	`InsertionSort` weist für kleine n geringere konstante Laufzeitfaktoren auf als `MergeSort`.

Bewertung: 1P pro korrektem Kreuz. Abzug bei falschen Kreuzen (min. 0P).

Aufgabe 5: AVL-Bäume (/4 Punkte)

Führen Sie die folgenden Operationen auf den gegebenen AVL-Bäumen aus.

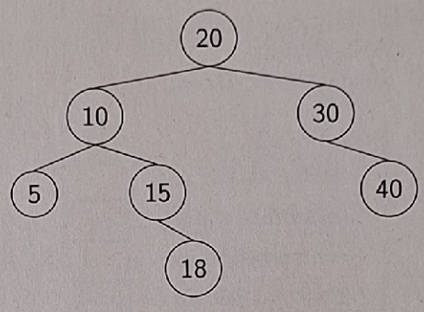
a.) Zeichnen Sie in der **linken Spalte** den Baum nach dem Einfügen/Löschen, aber **unmittelbar bevor** die Balancierungsfunktion `Balance(x)` aufgerufen wird.

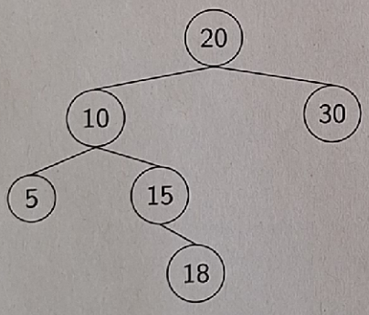
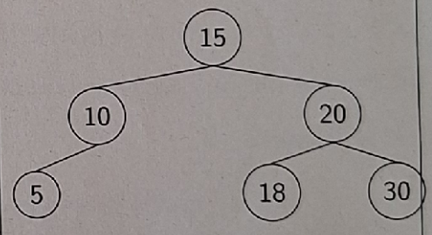
b.) Geben Sie in der **mittleren Spalte** den Zustand des Baumes vor `Balance(x)` an. Kreuzen Sie an, welche Rotation(en) notwendig sind und benennen Sie den Knoten, um den rotiert wird. Achten Sie auf die Reihenfolge.

c.) Zeichnen Sie in die **rechte Spalte** den resultierenden, balancierten AVL-Baum.

1. Operation: `Löschen(40)`

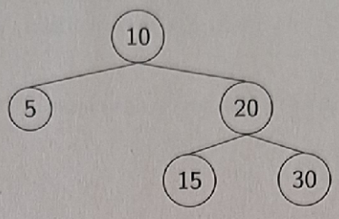
Gegebener Baum

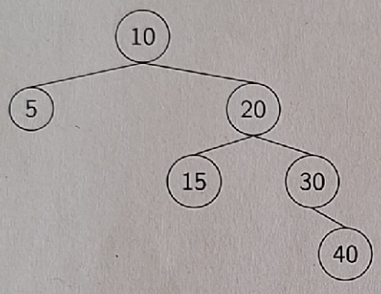
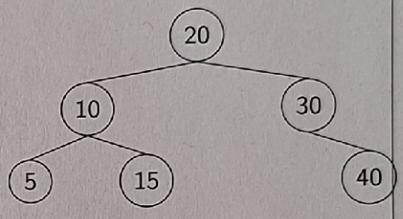


Baum vor Balance(x)	Maßnahmen	Resultierender Baum
	Der Baum vor Balance(x) ist ein ○ AVL-Baum. ⊗ Beinahe-AVL-Baum. ○ keine Rotation ○ 1. Rechtsrotation um ____ ⊗ 1. Linksrotation um 10 ⊗ 2. Rechtsrotation um 20 ○ 2. Linksrotation um ____	

2. Operation: `Einfügen(40)`

Gegebener Baum



Baum vor Balance(x)	Maßnahmen	Resultierender Baum
	Der Baum vor Balance(x) ist ein ○ AVL-Baum. ⊗ Beinahe-AVL-Baum. ○ keine Rotation ○ 1. Rechtsrotation um ⊗ 1. Linksrotation um **10** ○ 2. Rechtsrotation um ○ 2. Linksrotation um	

Bewertung (pro Tabelle):

- 0.5 Punkte für den korrekten Baum in der linken Spalte (vor Balance).
- 1.0 Punkt für die mittlere Spalte (0.5 für Status/Maßnahmen-Kreuz, 0.5 für korrekte Knoten).
- 0.5 Punkte für den korrekten resultierenden Baum in der rechten Spalte.

Gesamt: 4 Punkte

Aufgabe 6: Korrektheit von SelectionSort (/4 Punkte)

Gegeben: Ein Array A der Länge n mit Ganzzahlen. $A[x..y]$ bezeichnet das Teilarray von Index x bis Index y (beide Grenzen inklusive). SelectionSort ist wie folgt definiert (1-indiziert):

```
1 for i ← 1 to n - 1 do
2     min ← i
3     for k ← i + 1 to n do
4         if A[k] < A[min] then
5             min ← k
6     h ← A[min]
7     A[min] ← A[i]
8     A[i] ← h
```

Wir betrachten drei Invarianten:

- *Äußere Schleifeninvariante:* Zustand zu Beginn jeder Iteration der äußeren Schleife (Index i).
- *Innere Schleifeninvariante:* Zustand während der Suche nach dem Minimum in $A[i..n]$.
- *Endzustand*: Zustand nach Abschluss der äußeren Schleife.

Wählen Sie jeweils die korrekte Aussage.

(a) (1 Punkt) **Äußere Schleifeninvariante.**
Welche Aussage beschreibt eine zutreffende Invariante direkt vor Ausführung des Schleifenkörpers (Index i)?

◯ Alle Elemente in $A[1..i-1]$ sind größer als alle in $A[i..n]$.
◯ Der Bereich $A[i..n]$ ist bereits vollständig sortiert.
⊗ $A[1..i-1]$ enthält die $i-1$ kleinsten Werte von A in richtiger Reihenfolge.
◯ Für alle $j < i$ gilt $A[i] \leq A[j]$.

(b) (1 Punkt) **Innere Schleifeninvariante.**
Die innere Schleife läuft mit der Variable k von $i+1$ bis n und hält den Index des bisher gefundenen Minimums in `min`. Welche Invariante gilt nach jeder Iteration der inneren Schleife, d.h. nachdem die Elemente $A[i..k]$ verarbeitet wurden?

◯ A[`min`] ist der kleinste Wert im gesamten Array.
◯ Alle Werte in $A[k..n]$ sind größer als A[`min`].
◯ `min` bleibt immer gleich i, unabhängig von den Vergleichen.
⊗ A[`min`] ist der kleinste Wert aus $A[i..k]$.

(c) (1 Punkt) **Korrektheit nach Abschluss einer Iteration der äußeren Schleife.**
Nach dem Finden des Minimums in $A[i..n]$ und dem anschließenden Vertauschen – was ist garantiert korrekt?

◯ $A[1..i]$ enthält bereits alle n Elemente in endgültiger Reihenfolge.
◯ $A[i]$ ist größer als alle Elemente in $A[i+1..n]$.
◯ Das gesamte Array A ist sortiert.
⊗ $A[i]$ enthält den kleinsten Wert aus $A[i..n]$.

(d) (1 Punkt) **Endzustand nach Abschluss der äußeren Schleife.**
Welche Aussage stellt die finale Korrektheitsbedingung (Sortiertheit) dar?

◯ Für alle $1 \leq i < n$ gilt: $A[i] < A[i+1]$.
⊗ Für alle $1 \leq i < n$ gilt: $A[i] \leq A[i+1]$.
◯ Die Summe aller Elemente ist minimal.
◯ Das Minimum steht an Position n.