Lösung 03: Elementare Datenstrukturen

Aufgabe 1: Was macht das Programm?

a) Das Programm gibt die Binärdarstellung der Zahl *n* aus. Zunächst werden der Reihe nach das niedrigste Bit, dann das zweitniedrigste Bit, usw. auf den Stack gelegt. Anschließend wird über den Stack iteriert, wobei Iterieren hier bedeutet, dass wiederholt eine pop-Operation ausgeführt wird.
Beispiel: $n = 50 = (110010)_2$ in Binärdarstellung
In jeder Iteration wird das niedrigste Bit (*n* % 2) auf den Stack gelegt, anschließend wird durch den Ausdruck $n/2$ das niedrigste Bit „abgeschnitten“. Am Ende der while-Schleife sieht der Stack wie folgt aus:

1 <- Top-Zeiger
1
0
0
1
0 <- unten im Stack

Durch wiederholtes pop() ergibt sich der String 110010.

b) Der Code dreht die Reihenfolge der Elemente in der Queue um. Nach dem Ausführen sind alle Elemente in der umgekehrten Reihenfolge. Am besten sieht man das durch ein Beispiel.
- Die Queue enthalte zu Beginn die Elemente 1 2 3 4. Der Head ist 1, der Teil ist 4.
- Der Stack enthält nach der ersten while-Schleife die Elemente 1 2 3 4, wobei der Top-Zeiger auf der 1 steht.
- Durch die 2. while-Schleife werden alle Elemente des Stacks wieder in der umgekehrten Reihenfolge in die Queue eingetragen. Es ergibt sich 4 3 2 1, wobei nun der Head 4 ist und der Tail der Queue 1.

Aufgabe 2: Stack und Queue

Die Idee besteht darin, dass man bei jeder `dequeue` Operation zunächst alle aktuell auf dem Stack liegenden Elemente von *a* nach *b* legt. Auf *b* liegen die Elemente dann in der umgekehrten Reihenfolge als zuvor auf *a*. Man nimmt nun das oberste Element und schichtet zuletzt wieder alles zurück nach a.

```
boolean empty() {
   return a.empty();
}

void enqueue(Object o) {
   a.push(o);
}

Object dequeue() {
   while (!a.empty ()) {
       b.push(a.pop());
   }
   Object o = b.pop();
   while (!b.empty()) {
       a.push(b.pop());
   }
   return o;
}
```

Die (Worst-Case) Laufzeit für empty und enqueue ist jeweils $\Theta(1)$. Für dequeue muss man zweimal alle Elemente durchlaufen, die asymptotische Laufzeit beträgt $\Theta(n)$.

Alternatives (etwas besseres) Vorgehen: Eine Warteschlange kann man mit zwei Stacks implementieren, indem man den einen Stack `a` als Anfangs- und den anderen `b` als Endstück der Warteschlange benutzt. `Enqueue` legt das Element stets in O(1) auf a ab, `dequeue` nimmt sich immer das oberste Element aus b. Das geht so lange gut, wie in b noch Elemente vorhanden sind. Ist b jedoch leer, überträgt man alle Elemente von a auf b .

Aufgabe 3: Doppelt verkettete Ringliste

Lösung, siehe Quelltext im IntelliJ Projekt im GitLab: `Ring.java`.

Es ist **unbedingt notwendig(!),** die einzelnen Operationen mit Hilfe einer Skizze zu verdeutlichen.

Ein Vorteil dieser Implementierung einer Queue ist, dass man keinen eigenen Zeiger für `tail` und `head` halten muss. Entry zeigt immer auf den `head`, während der Eintrag danach (`head.next`) die `tail` der Queue ist.

Teilweise spielt es auch eine Rolle, in welcher Reihenfolge die „Zeiger“ gesetzt werden. Im Folgenden Beispiel darf man z.B. nicht die letzte Zeile vor der ersten Zeile ausführen.

```
entry.next.prev = newNode;
newNode.next = entry.next;
newNode.prev = entry;
entry.next = newNode;
```