Übung 03: Arrays, Verkettete Listen

Aufgabe 1: Listen und Arrays

Lösen Sie die Aufgaben wahlweise auf dem Papier mit Java-ähnlichem Pseudocode oder verwenden Sie die mitgelieferten Java-Klassen. Vorteil bei Letzterem: Sie können per JUnit testen.

a) `void reverseArray(int[] a):` Schreiben Sie eine Methode, die als Argument ein Integer-Array erhält und das gespiegelte Array zurückgibt. Sie dürfen **kein** temporäres Array erzeugen.

Gegeben ist eine einfache Java-Implementierung einer verketteten Liste.

b) `Node reverseIterative(Node first):`
Schreiben Sie eine **iterative** Methode, die den ersten Knoten einer verketteten Liste als Parameter erhält, dann die Liste umkehrt und den ersten Knoten der gespiegelten Liste als Ergebnis zurückgibt.

c) `Node reverseRecursive(Node first):`
Schreiben Sie eine **rekursive** Methode, die als Argument den ersten Knoten einer einfach verketten Liste nimmt, dann die Liste umkehrt und den ersten Knoten der gespiegelten Liste als Ergebnis zurückgibt.

```
public class ListReversal<Item>
         implements Iterable<Item>
{
  private Node first;
  public Node getFirst() {..}

  private class Node {
    Item item;
    Node next;
  }

  public void add(Item item) {..}
}
```

d) Recherchieren Sie: Gibt es in Python so etwas wie eine `ArrayList` oder eine `LinkedList` in Java? Wie heißen ggfs. die entsprechenden Datenstrukturen?

Aufgabe 2: Mergesort

Gegeben ist die eine *iterative* („Bottom-Up“) Implementierung von Mergesort.

a) Es wird das Array *a* = (5, 2, 4, 7, 1, 3, 6, 2) sortiert. Sie verwenden die *rekursiven* Variante aus der Vorlesung. Skizzieren Sie, wie der Algorithmus vorgeht. Geben Sie dazu z.B. den Zwischenstand des Gesamtarray nach jedem Aufruf der `merge`-Methode an.

b) Nun verwenden Sie die folgende *iterative* Variante. Wie sieht das Array jeweils am Ende einer Iteration der **äußeren** for-Schleife aus?

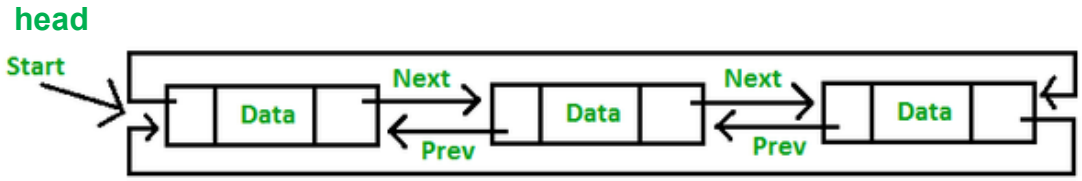
```
public static void sort(Comparable[] a) {
    int n = a.length;
    for (int len = 1; len < n; len *= 2) {
        for (int left = 0; left < n - len; left += 2 * len) {
            int middle  = left + len - 1;
            int right = Math.min(left + 2 * len - 1, n - 1);
            merge(a, left, middle, right); // merge of lecture
        }
    }
}
```

c) Diskutieren Sie für b) qualitativ die asymptotische Laufzeit? Gibt es einen Worst- und einen Best-Case?

d) Wie könnte man das Verfahren aus b) verbessern. Ziel soll sein, die Anzahl der Aufrufe der `merge`-Methode zu minieren. Erklären Sie die Strategie in 1-3 Sätzen.
Tipp: Für das folgende Beispiel-Array genügen 3 Aufrufe. *b* = (2, 16, 8, 9,10, 11, 12, 4, 5)

Aufgabe 3: Doppelt verkettete Ringliste

Eine *doppelt verkettete* Ringliste (*Circular Linked List*) ist eine doppelt verkettete Liste, bei dem kein `next-` und `prev`-Zeiger den Wert `null` hat. Alle Elemente sind in einem Kreis verbunden. Es genügt 1 Einstiegspunkt in die Datenstruktur, den wir `head` nennen.



Ihre Aufgabe: Sie implementieren

- `public void enqueue(Item item):` Einfügen in Queue. Das neue Element muss Vorgänger (`prev`) das aktuellen `head`-Zeigers werden.
- `public Item dequeue():` Entfernt das Element, auf das der `head`-Zeiger zeigt (=Head) und gibt das entfernte Element zurück (`null` falls Queue leer). Passen Sie den `head`-Zeiger an!

Wichtig: Der **`head`-Zeiger** zeigt nach jeder Operation auf den Knoten der Datenstruktur, das **als nächstes zu entnehmen ist (** = Head der Queue, „ältestes“ Element**).** Der **`next`-Zeiger** eines bestimmten Knotens zeigt auf den Knoten, der zeitlich gesehen unmittelbar **nach** diesem Knoten eingefügt wurde.

Zu beachten:

- Sie dürfen der Klasse `CircularLinkedList` keine weiteren Attribute hinzufügen.
- Fertigen Sie **Skizzen** an.
- Testen Sie mit der mitgelieferten `JUnit`-Testklasse.
- Achten Sie auf Randfälle, z.B. Datenstruktur hat aktuell nur 0 bzw. 1 Element.
- Passen Sie das Attribut `numElems` an, das speichert, wie viele Elemente aktuell gespeichert sind.
- Bei Problemen: **Debuggen!!!**