Theorie-Aufgaben (mit Stift auf Papier zu lösen)

Aufgabe 9.1
Die Methode merge(**int** [] a, **int** [] b, **int** [] erg), die **einen** merge-Schritt des MergeSort-Verfahrens implementiert, setzt zwar voraus, dass die beiden zu verschmelzenden Teil-Arrays a und b jeweils für sich sortiert sind, aber sie setzt nicht voraus, dass diese ungefähr gleich lang sind.

Geben Sie für die beiden Arrays **int** [] a = {1, 2, 3, 7, 9, 10, 11} und **int** [] b = {4, 5, 6, 8} an, welche Elemente miteinander verglichen werden, wenn die merge-Methode darauf angewendet wird.

Aufgabe 9.2
Sortieren Sie die Zahlenfolge

$$48 \quad 63 \quad 5 \quad 24 \quad 11 \quad 84 \quad 50 \quad 12 \quad 97 \quad 28 \quad 77 \quad 53 \quad 3$$

mit dem QuickSort-Verfahren.

Wählen Sie das mittlere Element bzw. das Element links von der Mitte als Pivot-Element.

Markieren Sie jeweils das Pivot-Element z.B. durch Unterstreichen und die zu sortierenden Teil-Arrays durch eckige Klammern $[,]$.
Markieren Sie außerdem die zu tauschenden Elemente.
Geben Sie nach jeder Vertauschung die getauschten Werte an. Gleichbleibende Elemente brauchen Sie **während eines Durchlaufs** nicht abschreiben - Sie sollten aber zur eigenen besseren Orientierung **nach jedem** Durchlauf das Array komplett angeben. Auf Arrays mit einer Länge ≤ 4 brauchen Sie **nicht** mehr das QuickSort-Verfahren anwenden. Sortieren Sie solche Arrays „durch Hinschauen“.

Programmieraufgaben

Aufgabe 9.3
Implementieren Sie in einer Klasse MengeUtil eine statische generische Methode min( ), die eine Menge eines „vergleichbaren“ Typs annimmt und das (gemäß der inneren Ordnung des Datentyps) kleinste Element der Menge zurückliefert.

Aufgabe 9.4
Implementieren Sie das MergeSort-Verfahren auf $T[\,]$ -Arrays *generisch*.

Aufgabe 9.5
Implementieren Sie das QuickSort-Verfahren auf $T[\,]$-Arrays *generisch*.

Aufgabe 9.6
Unter *binärer Suche* vesteht man das Prinzip, in einer Datenmenge, deren Elemente in geordeneter Reihenfolge vorliegen, den Suchwert zunächst mit einem Wert an „mittlerer“ Position zu vergleichen. Dann ist entweder der Suchwert gefunden, oder es kann entschieden werden, ob die Suche in dem „linken“ Teilbereich oder dem „rechten“ Teilbereich der Datenmenge fortgesetzt werden muss.

Die Sortierung kann man dadurch erreichen, dass bereits bei `insert (T e)` das neue Element an die in Sortierreihenfolge „richtige“ Position eingefügt wird. Das erhöht zwar den Zeitbedarf für das Einfügen, aber für die Suche nach einem Element kann der Zeitbedarf auf $\mathcal{O}(\log n)$ reduziert werden.

Definieren Sie eine Unterklasse SortedDynArray von DynArray, die das Prinzip der binären Suche anwendet, sodass die Methode **boolean** `contains (T e)` auf jeden Fall in Zeit $\mathcal{O}(\log n)$ arbeitet.

Im Material zu dieser Übung finden Sie einen JUnit-Test TimeTest mit dem Sie verifizieren können, wie sich das Laufzeitverhalten von `insert` bzw `contains` ändert.

Bei einem Test mit 10000 Einfüge- bzw Such-Operationen wurde folgende Werte ermittelt:

Einfügen in ein dynamischen Array: 0,004 sec Einfügen in ein sortiertes Array: 0, 095 sec

Suchen in einem dynamischen Array: 0,132 sec Suchen in einem sortierten Array: 0,008 sec

Im Material zu dieser Übung finden Sie außerdem die Definition des Interfaces SortedFolge.
Der Unterschied zum Interface Folge aus Übung 6 besteht einzig in der Signatur der Methode `insert` . Sie lautet jetzt **void** `insert (T e)`.

Implementieren Sie das Interface SortedFolge durch ein *sortiertes* dynamisches Array.

Überlegen Sie, inwieweit es sinnvoll ist, das Interface SortedFolge mittels eine (sortierten) EVL, EVLL oder DVL zu implementieren.