Übung 2 Abgabe 8.05.2019

Aufgabe 1 – Datenstruktur für Array, Stack und Queue 16 Punkte

Wir haben in der Vorlesung die abstrakten Containertypen *Array, Stack* und *Queue* kennengelernt. In dieser Aufgabe sollen Sie eine Python-Klasse UniversalContainer implementieren und testen, die die Fähigkeiten aller drei Container vereinigt. Die folgende Funktionalität soll unterstützt werden:

```
c = UniversalContainer()   # create empty container
c.size()                   # get number of elements
c.capacity()               # get number of available memory cells
c.push(v)                  # append element v at the end
c.popFirst()               # remove first element (Queue functionality)
c.popLast()                # remove last element (Stack functionality)
v = c[k]                   # read element at index k (Array functionality)
c[k] = v                   # replace element at k (Array functionality)
v = c.first()              # read first element (Queue functionality)
v = c.last()               # read last element (Stack functionality)
```

Die Klasse hat zu diesem Zweck einen internen Speicherbereich data_ der Größe capacity_, der aber nur bis zur Größe size_ gefüllt ist. Beim Aufruf c.push(v) wird Element v in die nächste freie Speicherzelle von data_ geschrieben und size_ inkrementiert. Gilt allerdings vor dem push, dass size_ == capacity_, muss man erst einen Speicherbereich mit größerer (z.B. verdoppelter) capacity_ schaffen und die vorhandenen Daten dahin umkopieren. Bei popLast() wird einfach size_ dekrementiert, so dass das vorletzte Element zum neuen letzten Element wird. Bei popFirst() werden alle Elemente einen Index heruntergeschoben und dadurch das erste überschrieben.

Der folgende Code gibt den Rahmen der Klasse vor, den Sie nur noch vervollständigen müssen:

```
class UniversalContainer:
    def __init__(self):       # constructor for empty container
        self.capacity_ = 1    # we reserve memory for at least one item
        self.data_ = [None]*self.capacity_   # the internal memory
        self.size_ = 0        # no item has been inserted yet
    def size(self):
        return self.size_
    def capacity(self):
        return ...                # your code here
    def push(self, item):             # add item at the end
        if self.capacity_ == self.size_:  # internal memory is full
            ...                   # your code to enlarge the memory
        self.data_[self.size_] = item
        self.size_ += 1
    def popFirst(self):
        if self.size_ == 0:
            raise RuntimeError("popFirst() on empty container")
        ...                       # your code here
    def popLast(self):
        if self.size_ == 0:
            raise RuntimeError("popLast() on empty container")
        ...                       # your code here
    def __getitem__(self, index):     # __getitem__ implements v = c[index]
```

```
        if index < 0 or index >= self.size_:
            raise RuntimeError("index out of range")
        return ...                  # your code here
    def __setitem__(self, index, v):  # __setitem__ implements c[index] = v
        if index < 0 or index >= self.size_:
            raise RuntimeError("index out of range")
        ...                         # your code here
    def first(self):
        return ...                  # your code here
    def last(self):
        return ...                  # your code here
```

Wir haben in der Vorlesung gelernt, dass die Datenstruktur folgende Axiome erfüllt:

- Ein neuer Container hat die Größe 0.
- Zu jeder Zeit gilt: `c.size() <= c.capacity()`.
- Nach einem push gilt: (i) die Größe hat sich um eins erhöht, (ii) das gerade eingefügte Element ist jetzt das letzte, (iii) alle anderen Elemente haben sich nicht verändert, (iv) wenn der Container vorher leer war, ist das eingefügte Element auch das erste, andernfalls bleibt das erste Element unverändert, (v) ein nachfolgendes `popLast()` reproduziert wieder den Container vor dem push (Sie können für diesen Test mit der Funktion `deepcopy()` aus dem Modul `copy` eine Kopie des ursprünglichen Containers erzeugen).
- Nach `c[k] = v` gilt: (i) die Größe bleibt unverändert, (ii) Index `k` enthält das Element `v`, (iii) die übrigen Elemente haben sich nicht verändert.
- Nach `c.popLast()` gilt: (i) die Größe hat sich um eins verringert, (ii) die Elemente vom ersten bis zum vorletzten bleiben unverändert.
- Nach `c.popFirst()` gilt: (i) die Größe hat sich um eins verringert, (ii) das zweite bis letzte Element sind einen Index nach unten gerückt.
- Wenn der Container nicht leer ist, gilt stets (i) `c.first() == c[0]` und (ii) `c.last() == c[c.size()-1]`.

Aufgaben (geben Sie Ihre Lösung im File `universal_container.py` ab):

a) Vervollständigen Sie die Implementierung von `UniversalContainer`.

b) Schreiben Sie eine Funktion `testContainer()`, die für verschiedene repräsentative Fälle mittels `assert` (z.B. '`assert c.size() == 0`') testet, dass die obigen Axiome gelten.

Aufgabe 2 – Eigenschaften der Sortierung 8 Punkte

a) Um Daten sortieren zu können, muss auf den Elementen eine Relation '≤' definiert sein, die die Eigenschaften einer *totalen Ordnung* (siehe https://de.wikipedia.org/wiki/Ordnungsrelation) hat. Zeigen Sie, dass dadurch auch die anderen Relationen '<', '>', '≥', '==' und '!=' festgelegt sind, indem Sie diese so definieren, dass in den Definitionen nur die '≤' Relation und logische Operationen (not, and, or) vorkommen.

b) Nach dem Sortieren eines Arrays `a` der Größe n ist das folgende Axiom erfüllt: Für alle Paare von Indizes j und k mit $j < k$ sind die zugehörigen Arrayelemente sortiert, d.h. es gilt $a[j] \leq a[k]$. Das sind $n(n-1)/2$ Bedingungen (eine für jedes mögliche Paar $j < k$), die man alle prüfen muss, um die Korrektheit der Sortierung nachzuweisen. Zeigen Sie mit Hilfe der Eigenschaften der '≤' Relation, dass dieser Aufwand unnötig ist und man auch mit $(n-1)$ Tests

auskommt. Welche Tests sollte man ausführen, und warum sind die übrigen dann automatisch erfüllt?

Aufgabe 3 – Vergleichen und Testen von Sortierverfahren 20 Punkte

a) Implementieren Sie die Algorithmen insertion sort, merge sort und quick sort aus der Vorlesung (Abgabe als File "sort.py"). Fügen Sie dabei Zählvariablen ein, die bei jedem Aufruf die Anzahl der Vergleiche zwischen Arrayelementen während des Sortierens zählen: *8 Punkte*

```
>>> a1 = [...]                        # the array to be sorted
>>> a2 = copy.deepcopy(a1)                  # a copy of the array
>>> a3 = copy.deepcopy(a1)                  # another copy of the array
>>> a1, comparisonCount1 = insertionSort(a1) # return sorted array
                                  # and comparison counter[1]
>>> a2, comparisonCount2 = mergeSort(a2)     # likewise
>>> a3, comparisonCount3 = quickSort(a3)     # likewise
```

Messen Sie für die drei Algorithmen die Anzahl der Vergleiche in Abhängigkeit von der Arraygröße und stellen Sie die Ergebnisse in einem Diagramm dar.[2] Suchen Sie geeignete Konstanten *a...i*, so dass die Kurven durch Funktionen

$a\,N^2+b\,N+c$ (insertion sort) $d\,N\log N+e\,N+f$ (merge sort)

$g\,N\log N+h\,N+i$ (quick sort)

gut approximiert werden (die Fits müssen nicht sehr genau sein).

b) Messen Sie mit dem timeit-Modul die Laufzeit der drei Algorithmen (wieder in Anhängigkeit von der Arraygröße), stellen Sie die Ergebnisse graphisch dar und vergleichen Sie mit den Ergebnissen von a). Ist die funktionale Form aus a) – mit anderen Konstanten *a...i* – auch für diese Messung geeignet? *6 Punkte*

c) Wir haben in der Vorlesung (am 29.4.2017) drei Nachbedingungen für die Korrektheit eines Sortieralgorithmus behandelt: die Arrays müssen davor und danach die gleiche Größe haben, die gleichen Elemente enthalten, und das Ergebnis muss sortiert sein. Entwickeln Sie einen Algorithmus, der diese Bedingungen prüft (dieser Algorithmus muss nicht effizient sein), und begründen Sie dessen Vorgehen. Denken Sie dabei daran, dass Zahlen mehrfach vorkommen können ([3,2,3,1]≠[1,1,2,3] ist ein Fehler!). Implementieren Sie den Algorithmus als Pythonfunktion *6 Punkte*

```
>>> correct = checkSorting(arrayBefore, arrayAfter)  # return True or False
```

und geben Sie die Implementation ebenfalls in "sort.py" ab.

Hinweis: über die Verwendung der timeit-Bibliothek kann man hier
http://alda.iwr.uni-heidelberg.de/index.php/Sortieren
nachlesen.

Bitte laden Sie Ihre Lösung bis zum 08.05.2019 um 12:00 Uhr auf Moodle hoch.

---

[1] Eine Python-Funktion kann mehrere Werte zurückgeben, indem man "return r1, r2" schreibt.

[2] Die Wahl des Werkzeugs zur Erstellung von Diagrammen ist freigestellt. Wer z.B. MS Excel oder Matlab beherrscht, kann dies verwenden. Handarbeit auf Millimeterpapier ist ebenso zulässig. Der Standard unter Python ist matplotlib (download unter matplotlib.sourceforge.net), das ich deshalb in erster Linie empfehle. Ebenfalls gut und frei erhältlich ist Gnuplot (www.gnuplot.info). Man übergibt hier die zu zeichnenden Daten in Form von Textfiles, die man zuvor in Python erstellt, oder man benutzt das Modul gnuplot.py (gnuplot-py.sourceforge.net) für eine direkte Pythoneinbindung.