A computer is deemed Reliable when its users are never surprised by something its designers must later apologize for. – W. Kahan

Nr.	Datum	Inhalt
1	9.4.	Organisatorisches, Scheinbedingungen, Credit points Einführung: Definition von Algorithmen und Datenstrukturen, Geschichte Fundamentale Algorithmen (create, destroy, assign, swap, copy, identity, equality) (call by) value vs. reference, deep vs. shallow copy/compare Fundamentale Datenstrukturen: Zahlen und Container
Ü 1		Python-Tutorial Sieb des Eratosthenes Wert- und Referenzsemantik
2	10.4.	Anforderungen der Algorithmen an Container Einteilung der Container Listen, Arrays, Stacks und Queues
3	16.4.	Sortieralgorithmen: selection sort, merge sort Zeitmessung in Python
Ü 2		Sortieren: Implementation und Geschwindigkeitsvergleich (Kurven in Abhängigkeit von Problemgröße) Entwicklung eines effizienten Algorithmus: Bruchfestigkeit von Gläsern
4	17.4.	Quicksort und seine Varianten
5	23.4.	Korrektheit, Korrektheitsbeweise Programming by contract, Vor- und Nachbedingungen, Invarianten Testen, Unit Tests, Execution paths
Ü 3		Experimente zur Effektivität von Unit Tests Deque-Datenstruktur: Vor- und Nachbedingungen der Operationen, Implementation und Unit Tests
6	24.4.	Exceptions und Exception Handling Geschwindigkeit und Optimierung: Innere Schleife, Caches, locality of reference Komplexität versus Geschwindigkeit
7	30.4.	Komplexitätsklassen Bester, schlechtester, durchschnittlicher Fall Amortisierte Komplexität
Ü 4		Theoretische Aufgaben zur Komplexität Amortisierte Komplexität von array.append() Optimierung der Matrizenmultiplikation
	1.5.	*Himmelfahrt*
8	7.5.	Suchen: Binäre Suche, Suchbäume, balancierte Bäume Komplexität der Suche
Ü 5		Implementation und Analyse eines Binärbaumes Anwendung: einfacher Taschenrechner
9	8.5.	Erhaltung der Balance beim Einfügen, Rotationen, selbst-balancierende Bäume
10	14.5.	Prioritätswarteschlange, Heap
Ü 6		Treap-Datenstruktur: Verbindung von Suchbaum und Heap
11	15.5.	Mengen Hashing Dictionaries als Baum und als Hashtabelle
12	21.5.	Iteration vs. Rekursion Typen der Rekursion und ihre Umwandlung in Iteration Iteratoren
Ü 7		Übungen zu Rekursion und Iteration: Fakultät, Koch-Schneeflocke, Auflösung rekursiver Formeln, Umwandlung von Rekursion in Iteration
	22.5.	*Fronleichnam*
13	28.5.	Abstrakte Datentypen Generizität: Required Interface, Adapter und Typattribute Funktoren (Beispiel: Ordnung für Sortieren)
Ü 8		Übungen zur Generizität: Sortieren mit veränderter Ordnung, Adapterklassen
14	29.5.	Zahlendatentypen und Arithmetik, Byteorder, IEEE floating point Algebraische Konzepte Operator Overloading, Anwender-definierte Zahlentypen (z.B mit physikalischen Einheiten)

15	4.6.	Graphen und Graphendatenstrukturen Adjazenzlisten und Adjazenzmatrizen Gerichtete und ungerichtete Graphen Vollständiger Graph Pfade, Zyklen
Ü 9		Elementare Graphenaufgaben (Adjazenzlisten aufschreiben, Beweis e <= 3n-6 für planare Graphen, Zyklen eines vollständigen Graphen) Implementation einer Graphklasse Iteratoren für Tiefensuche und Breitensuche
16	5.6.	Tiefensuche und Breitensuche Zusammenhang, mehrfacher Zusammenhang, Komponenten
17	11.6.	Gewichtete Graphen Minimaler Spannbaum
Ü 10		Anwendung: Weg aus einem Labyrinth Anwendung: Erzeugen einer perfekten Hashfunktion
18	12.6.	Best-first search (Dijkstra) Kürzeste Wege Most-Promising-first search (A*)
19	18.6.	Prinzipien des Algorithmenentwurfs Repetition Orthogonale Zerlegung Divide and Conquer (inklusive branch-and-bound) Randomisierung Optimierung, Zielfunktionen Systematisierung von Algorithmen aus der bisherigen Vorlesung
Ü 11		Anwendung: Routenplaner Beispiele für Divide and Conquer: pow-Funktion Beispiel für Methode der kleinsten Quadrate: Approximation von Kreisen
20	19.6.	Analytische Optimierung: Methode der kleinsten Quadrate, Approximation von Geraden
21	25.6.	Zufallszahlen, Zyklenlänge, Pitfalls Zufallsverteilungen, Box-Muller Transformation Randomisierte vs. deterministische Algorithmen Las Vegas vs. Monte Carlo Algorithmen Beispiel für Las Vegas: Randomisiertes Quicksort
Ü 12		Naiver (deterministischer) und randomisierter Primzahltest, Laufzeitvergleich RANSAC für Kreise
22	26.6.	Beispiel für Monte Carlo: randomisierte Integration, randomisierter Primzahltest RANSAC-Algorithmus, Erfolgswahrscheinlichkeit Beispiel: Linien finden (Deterministisch vs. RANSAC)
23	2.7.	Greedy-Algorithmen, Bedingung für Optimalität Beispiele für Greedy-Algorithmen
Ü 13		Theoretische und praktische Aufgaben zur dynamische Programmierung
24	3.7.	Dynamische Programmierung (1)
25	9.7.	Dynamische Programmierung (2)
Ü 14		Sudoku-Löser
26	10.7.	Exhaustive Search, Backtracking NP-Vollständigkeit, Algorithmen mit exponentieller Laufzeit
27	16.7.	Approximation bei NP-vollständigen Problemen
28	17.7.	Quantum computing

Übung 1 Abgabe 17.4.2008

Aufgabe 1 – Einführung in Python 10 Punkte

a) Installieren Sie die Programmiersprache Python auf Ihrem System (siehe www.python.org). Wenn Sie Linux verwenden, ist Python wahrscheinlich bereits installiert.

b) Arbeiten Sie aus dem Python-Tutorium (siehe docs.python.org/tut/tut.html) die Abschnitte 1 bis 7 durch. Sie sollen dabei die angegebenen Beispiele auf Ihrer Maschine nachvollziehen. Die folgenden Unterabschnitte (die sich an fortgeschrittene Programmierer richten) können übersprungen werden: 2.2, 4.7, 5.1.3, 5.1.4, 5.6, 5.7, 5.8, 6.4.

c) Implementieren Sie das Sieb des Eratosthenes zur Bestimmung von Primzahlen (den Algorithmus finden Sie im Internet). Geben Sie Ihre Lösung in einem File „sieve.py“ ab, das wie folgt benutzt werden kann:

```
>>> execfile("sieve.py") # load the function sieve()
>>> primes = sieve(1000) # return an array of all primes below 1000
```

Aufgabe 2 – Werte und Referenzen 10 Punkte

a) Wenn Sie im Internet auf eine interessante Seite stoßen, die Sie sich merken wollen, können Sie entweder ein Lesezeichen setzen (bookmark), oder die Seite abspeichern (save page as). Dies entspricht genau der Unterscheidung zwischen Referenzen und Werten in der Programmierung. Beschreiben Sie Vor- und Nachteile beider Vorgehensweisen. Denken Sie dabei z.B. an Speicherverbrauch, Offline-Betrieb, Datenkonsistenz und die Möglichkeit, dass die Originalseite aktualisiert oder gelöscht werden könnte.

b) In Python erfolgen Zuweisungen standardmäßig per Referenz.[1] Für alle Datentypen kann man eine Zuweisung per Wert erzwingen, wenn man das Python-Modul "copy" verwendet. Informieren Sie sich über dieses Modul (siehe docs.python.org/lib/module-copy.html). Analog gilt für Vergleiche, dass der operator "==" (z.B. `a == b`) die Werte von zwei Variablen vergleicht, während der operator "`is`" (z.B. `a is b`) prüft, ob zwei Variablen dasselbe Objekt referenzieren. Führen Sie folgendes Experiment durch:

```
>>> import copy
>>>
>>> a = [1, 2, 3, 4]
>>> b = a
>>> c = copy.deepcopy(a)

>>> print a == b, a == c, b == c   # are they equal?
>>> print a is b, a is c, b is c   # are they identical?

>>> a[0] = 42                      # change array a
>>> print a == b, a == c, b == c   # are they still equal?
>>> print a is b, a is c, b is c   # are they identical?
>>> print a[0], b[0], c[0]         # which values do you get?
```

Beschreiben und begründen Sie, wie sich das Programm verhält.

[1] Ausnahmen bilden die Zahlen (Typen `bool`, `int` und `float`), die stets als Werte zugewiesen werden.

Übung 2 Abgabe 24.4.2008

Aufgabe 1 – Vergleichen und Testen von Sortierverfahren ? Punkte

a) Implementieren Sie die Algorithmen selection sort und merge sort aus der Vorlesung (Abgabe als File "`sort.py`"). Fügen Sie dabei eine Zählvariable ein, die bei jedem Aufruf zählt, wie viele Vergleiche zwischen Arrayelementen während des Sortierens ausgeführt wurden:

```
>>> a1 = [...]                              # the array to be sorted
>>> a2 = copy.deepcopy(a1)                  # a copy of the array
>>> comparisonCount = selectionSort(a1) # sort in-place and return counter
>>> comparisonCount = mergeSort(a2)     # likewise
```

Messen Sie für beide 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...f*, so dass die Kurven durch Funktionen

$$a\,N^2 + b\,N + c \quad \text{(selection sort)}$$
$$d\,N \log N + e\,N + f \quad \text{(merge sort)}$$

möglichst gut approximiert werden.

b) Die Laufzeit von Algorithmen kann in Python mit Hilfe des "timeit"-Moduls gemessen werden (siehe docs.python.org/lib/module-timeit.html). Messen Sie mit diesem Modul die Laufzeit von selection sort und merge sort (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...f* – auch für diese Messung geeignet?

c) Wir haben in der Vorlesung 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

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

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

Aufgabe 2 – Entwicklung eines effizienten Algorithmus ? Punkte

Gläser gehen in Gaststätten häufig zu Bruch. Deshalb werden seit Jahren verbesserte Glassorten entwickelt. Um die Stabilität der neuen Gläser zu überprüfen, wird folgendes Experiment durchgeführt: Es steht ein 3 m hoher Ständer zur Verfügung, an dem im Abstand von 10 cm Plattformen befestigt sind. Allgemein nehmen wir an, dass *K* Höhen überprüft werden können. Es soll die größte Höhe bestimmt werden, aus der ein Glas ohne Beschädigung fallen kann. (Wir lassen hier außer acht, dass der Versuch aus statistischen Gründen mehrmals wiederholt werden sollte.)

a) Wie geht man vor, wenn von jedem Glastyp nur ein Exemplar zur Verfügung steht? Wie viele Schritte (als Funktion von *K*) werden im ungünstigsten Fall benötigt (mit Begründung)?

b) Diesmal stehen von jedem Glas zwei Exemplare zur Verfügung. Kann man durch geschicktes Vorgehen jetzt schneller zum Ziel kommen (d.h. mit weniger Schritten im ungünstigsten Fall), und wenn ja, wie?

c) Wie sieht es aus, wenn von jedem Typ *N*>2 Exemplare vorhanden sind?

---

[2] Die Wahl des Werkzeugs zur Erstellung von Diagrammen ist freigestellt. Wer z.B. MS Excel oder Matlab beherrscht, kann dies verwenden. Handarbeit ist ebenfalls zulässig. Gut und frei erhältlich ist Gnuplot (www.gnuplot.info). Man übergibt hier die zu zeichnenden Daten in Form eines Textfiles, das man zuvor in Python erstellt. Eine noch bessere Integration mit Python wird durch die Installation des Moduls gnuplot.py (gnuplot-py.sourceforge.net) erreicht.

Übung 3 Abgabe 2.5.2008

Aufgabe 1 – Korrektheit ? Punkte

a) In der Vorlesung haben wir die Division durch sukzessives Subtrahieren behandelt, um Vorbedingungen, Nachbedingungen und Invarianten zu erläutern und einen formalen Korrektheitsbeweis exemplarisch vorzuführen:

```
Vorbedingungen: x > 0, y > 0
q = 0
r = x
while y <= r:
    r = r - y
    q = q + 1
Nachbedingungen: y > r
Invarianten:     x_i == x and y_i == y and x == r_i + y * q_i
```

(*q* ist der Quotient, *r* der Rest von *x* / *y*, das Subskript *i* bezeichnet die Werte der Variablen nach dem *i*-ten Durchlauf durch die Schleife). Erstellen Sie per Hand eine Tabelle, die die Werte der Variablen nach dem *i*-ten Schritt auflistet und die Invarianten überprüft (eine solche Tabelle ist eine langweilige, aber sehr effektive Methode zum Debuggen von Algorithmen):

Schritt	x	y	q	r	r + y*q
0	31	9	0	31	31+9*0 == 31
1	...				
...					

b) Wir haben gezeigt, dass sich der Algorithmus von Freivalds zum Testen der Matrixmultiplikation eignet, weil die Wahrscheinlichkeit, einen vorhandenen Fehler trotz wiederholter Anwendung des Tests nicht zu entdecken, exponentiell (mit $1/2^K$) sinkt. Der Algorithmus lautet:

```
Eingabe: NxN-Matrizen A, B, C, von denen behauptet wird, dass C = A*B
         Anzahl K der Wiederholungen
1. Für i = 0...K-1:
   a. wähle einen Vektor u der Länge N mit zufälligen Nullen und Einsen
   b. berechne die Vektoren v = A*(B*u) und w = C*u (beachte die Klammern!)
   c. falls v != w: return False     # Fehler gefunden
2. return True                       # keinen Fehler gefunden
```

Implementieren Sie die Matrixmultiplikation und den Algorithmus von Freivalds.[3] Zur Erzeugung von Zufallszahlen verwenden Sie das Python-Modul "random" (siehe docs.python.org/lib/module-random.html). Bauen Sie jetzt absichtlich Fehler in die Matrixmultiplikation ein (verändern Sie ein zufälliges Element von A oder C, brechen Sie eine Schleife einen Schritt zu früh ab, und dergleichen – beschreiben Sie, was Sie gemacht haben) und lassen Sie den Testalgorithmus für verschiedene *K* laufen. Erstellen Sie eine Tabelle oder ein Diagramm, wie häufig der Fehler für jedes *K nicht* gefunden wurde. Wird der erwartete exponentielle Abfall der Häufigkeiten bestätigt?

c) Eine double-ended Queue (Deque) vereinigt die Fähigkeiten einer Queue (first in – first out) mit denen eines Stacks (last in – first out). Wenn die Deque eine feste Maximalgröße hat, kann sie mit Hilfe eines Arrays implementiert werden, das als Ringspeicher arbeitet: Die Funktion `push()` füllt das Array von vorn nach hinten, die Funktionen `popFirst()` und `popLast()` leeren es wieder in der Queue- bzw. Stackreihenfolge. Der gerade aktive Bereich des Arrays wird durch einen Anfangs- und einen Endindex bezeichnet. Wenn diese Indizes beim Einfügen

[3] Matrizen sollen hier durch eindimensionale Arrays implementiert werden, so dass z.B. das Matrixelement C_{ij} als `C[i+j*N]` adressiert wird.

oder Auslesen die Arraygrenzen überschreiten, werden sie zyklisch behandelt (d.h. max+1→0, 0-1→max – dies realisiert den "Ring"). Natürlich darf der Endindex nie den Anfangsindex "überrunden".

I. Geben Sie Vor- und Nachbedingungen für die Funktionen `q=create(N)`, `size(q)`, `capacity(q)`, `push(q, x)`, `x=popFirst(q)` und `x=popLast(q)` vollständig an (N ist dabei die Maximalgröße, `capacity(q)` gibt die Maximalgröße zurück, `size(q)` ist die aktuelle Größe, `x` das einzufügende bzw. ausgelesene Element).

II. Implementieren Sie die Datenstruktur in Python. Achten Sie dabei darauf, dass jede Funktion ihre Vorbedingungen prüft und eine Exception vom Typ `RuntimeError` auslöst, falls diese nicht erfüllt sind. Die Fehlermeldung sollte klar ausdrücken, was der Nutzer tun muss, um den Fehler zu vermeiden.

III. Informieren Sie sich über das Python-Modul "unittest" (docs.python.org/lib/module-unittest.html). Implementieren Sie geeignete Unit Tests für Ihre Datenstruktur. Testen Sie dabei die Nachbedingungen im typischen Fall (teilweise gefüllte Queue), die Randfälle (leere Queue, fast volle Queue), sowie die Fehlerfälle (d.h. ob jeweils die erwarteten Exceptions ausgelöst werden, wenn Vorbedingungen nicht erfüllt sind). Achten Sie darauf, dass sämtliche Teile des Codes getestet werden ("complete code coverage").

Implementation und Tests sollen im File "`deque.py`" abgegeben werden.

Übung 4 Abgabe 8.5.2008

Aufgabe 1 – Komplexität und Geschwindigkeit ? Punkte

a) Beweisen Sie die Vereinfachungsregeln für Ausdrücke in *O*-Notation:

$$\begin{aligned} f(x) &\rightarrow O\big(f(x)\big) \\ O\big(c\, f(x)\big) &\rightarrow O\big(f(x)\big) \\ O\big(f(x)\big)\, O\big(g(x)\big) &\rightarrow O\big(f(x)g(x)\big) \\ O\big(f(x)\big) + O\big(g(x)\big) &\rightarrow O\big(f(x)\big) \text{ falls } g(x) = O\big(f(x)\big) \end{aligned}$$

b) Ordnen Sie die folgenden Funktionen nach aufsteigender Komplexität:

$$\begin{aligned} f_1(x) &= 3^x \\ f_2(x) &= \sqrt{x} + \log x \\ f_3(x) &= x^x \\ f_4(x) &= x \log x \\ f_5(x) &= 2^{\sqrt{\log x}} \\ f_6(x) &= x^3 + 12x^2 + 200x + 999 \end{aligned}$$

und begründen Sie Ihre Entscheidung.

c) Gegeben seien drei Algorithmen mit Laufzeit

$$\begin{aligned} a_1(N) &= N^2 + N + 10 \\ a_2(N) &= 15\, N \log_2 N \\ a_3(N) &= 2^N \end{aligned}$$

(jeweils für $N > 1$). Geben Sie Intervalle für N an, wo jeder der drei Algorithmen am schnellsten abläuft. Ordnen Sie die Algorithmen außerdem nach ihrer Komplexität. In welchem Intervall stimmt diese Ordnung mit der Ordnung nach Laufzeit überein?

d) Beweisen Sie, dass das Sieb des Eratosthenes (siehe Übung 1) in $O(N \log\log N)$ liegt (wobei N die größte betrachtete Zahl ist). Benutzen Sie dazu die Formel $\sum_{p \le N} \frac{1}{p} = O(\log\log N)$, wobei die Summe sich über die Primzahlen p bis maximal N erstreckt. Bauen Sie in den Code aus Übung 1 (`"sieve.py"`) eine Zählervariable ein, die die Gesamtzahl der Iterationen der inneren Schleife zählt, und überprüfen Sie experimentell die Korrektheit des obigen Resultats.

e) Überprüfen Sie experimentell, ob das Python-Array (Typ `"list"`) ein dynamisches Array ist, d.h. ob die amortisierten Kosten von `array.append(x)` konstant sind. Benutzen Sie dafür das "timeit"-Modul (siehe Übung 2).

Bonusaufgabe: Wer die Programmiersprache C beherrscht, kann zusätzlich den Quellcode von Python anschauen und die relevanten Teile der Implementation von `array.append(x)` kurz erläutern. Der Code befindet sich in der Python-source distribution in der Funktion `"PyList_Append()"` im File `"objects/listobject.c"`.

f) Eine naive Implementation der Multiplikation von zwei `size*size` Matrizen lautet[4]

```
for i in range(size):
    for j in range(size):
        for k in range(size):
            C[i+j*size] += A[i+k*size]*B[k+j*size]
```

Diese Implementation ist relativ langsam, weil die innere Schleife redundante Berechnungen enthält und der Cache schlecht ausgenutzt wird. Optimieren Sie die Implementation (z.B. durch Umstellung der Reihenfolge der Schleifen, Verschiebung von invarianten Teilausdrücken), begründen Sie Ihre Optimierungen und messen Sie die Verbesserung für `size=100`. (Verwenden Sie wieder das "timeit"-Modul. Die Laufzeit sollte sich ungefähr halbieren.) Ändert sich durch die Optimierung die Komplexität des Algorithmus?

[4] Matrizen sind wieder durch eindimensionale Arrays implementiert (siehe Übung 3). Wir nehmen an, dass alle C_{ij} bereits auf 0 initialisiert wurden.

Übung 5 Abgabe 15.5.2008

Aufgabe 1 – Binärbäume ? Punkte

a) Gegeben sei eine Python-Klasse[5] `Node` mit folgender Definition

```
class Node:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None
```

Alle Knoten eines Baumes sind vom Typ `Node`, ein (Teil-)Baum ist durch seine Wurzel repräsentiert (`None` bezeichnet einen leeren Teilbaum). Implementieren Sie die in der Vorlesung behandelten Funktionen und entsprechende Unit Tests:

`treeInsert(rootnode,key)`: neuen Schlüssel in den Baum einfügen (falls der Schlüssel bereits enthalten war, soll der Baum nicht verändert werden),

`treeRemove(rootnode,key)`: angegebenen Schlüssel entfernen (falls der Schlüssel nicht vorhanden ist, soll eine Exception ausgelöst werden) ,

`treeHasKey(rootnode,key)`: gibt `None` zurück, wenn der Schlüssel nicht im Baum vorhanden ist, oder den `Node`, der diesen Schlüssel enthält.

b) Entwickeln Sie einen Algorithmus, der die Tiefe des Baumes (den maximalen Abstand von der Wurzel zu einem Blatt) bestimmt und implementieren Sie ihn als Funktion `depth=treeDepth(rootnode)`. Konstruieren Sie Eingaben, die die Tiefe minimieren bzw. maximieren (günstigste und ungünstigste Reihenfolge von `treeInsert`-Aufrufen) und prüfen Sie, dass die erwartete Tiefe in der Tat erreicht wird.

c) Angenommen, Sie können die Eingabeschlüssel vorsortieren. Wie gehen Sie vor, damit der Baum nach dem Einfügen der Schlüssel eine möglichst geringe Tiefe hat?

d) Beweisen oder widerlegen Sie folgende Aussage: Wenn aus einem Binärbaum erst der Schlüssel X und danach der Schlüssel Y entfernt wird, entsteht der gleiche Baum wie bei umgekehrter Reihenfolge.

Aufgabe 2 – Taschenrechner ? Punkte

Wir wollen Ausdrücke der Form "2+5*3" oder "2*4*(3+(4-7)*8)-(1-6)" auswerten, die als Zeichenketten gegeben sind.[6] Es gelten die üblichen Rechenregeln (Klammern habe die höchste Priorität, Punktrechnung geht vor Strichrechnung). Zur Vereinfachung können Sie annehmen, dass nur einstellige Zahlen vorkommen.

Operationen mit gleicher Priorität werden von links nach rechts ausgewertet. Trifft man jedoch auf eine Klammer bzw. eine Operation höherer Priorität, muss man den Substring bis zur zugehörigen schließenden Klammer bzw. bis zum nächsten Operator mit niedrigerer Priorität suchen und die Auswertung rekursiv auf diesen Substring anwenden. Dadurch ergibt sich ein Binärbaum.

a) Entwickeln Sie einen Algorithmus, der den zu einem Ausdruck korrespondierenden Binärbaum aufbaut, wobei jeder innere Knoten einen Operator ('+', '-', '*', '/') repräsentiert, jeder Unterbaum einen linken bzw. rechten Operanden, und jedes Blatt eine Zahl. Begründen und implementieren Sie diesen Algorithmus.

b) Skizzieren Sie die Bäume, die sich für obige Beispiele ergeben.

c) Implementieren Sie ein Verfahren, um einen Ausdruck mit Hilfe des erstellten Baums auszurechnen.

d) Schreiben Sie Unit Tests für Ihr Verfahren.

---

[5] Lesen Sie die Abschnitte 9.3 und 9.4 des Python-Tutoriums zum Thema Klassen.

[6] Die Verwendung der Python-Funktionen `eval()` bzw. `exec` ist in dieser Aufgabe nicht erlaubt.

Übung 6 Abgabe 23.5.2008

Aufgabe 1 – Treaps ? Punkte

Bei selbstbalancierenden Bäumen versucht man, die Höhe des Baumes zu minimieren. Dies ist aber gar nicht das eigentliche Ziel der Optimierung: In Wirklichkeit will man die Zugriffszeit auf die Elemente minimieren. Wenn die Schlüssel mit sehr unterschiedlicher Häufigkeit abgefragt werden, ist ein balancierter Baum dafür nicht die optimale Lösung. Stattdessen will man häufige Schlüssel nahe der Wurzel des Baumes abspeichern, auch wenn seltene Schlüssel dadurch in tieferen Ebenen landen als beim balancierten Baum. Dies wird durch den sogenannten Treap erreicht, der die Eigenschaften von Suchbäumen und Heaps verbindet. Neben dem Schlüssel hat hier jedes Element auch eine Priorität, und Elemente mit hoher Priorität liegen nahe der Wurzel des Baumes. Dies erreicht man, indem jeder Teilbaum zwei Eigenschaften erfüllt:

(1) Sortierung nach Schlüsseln: Alle Elemente im linken Teilbaum haben kleinere Schlüssel als die Wurzel des Teilbaums, alle Elemente im rechten Teilbaum größere.

(2) Sortierung nach Prioritäten: Kein Element im linken oder rechten Teilbaum hat höhere Priorität als die Wurzel des Teilbaums.

Die Erfinder der Treap-Datenstruktur haben gezeigt, dass beide Eigenschaften gleichzeitig erfüllbar sind. Der grundlegende Einfügealgorithmus lautet:

1. Füge ein neues Element entsprechend seines Schlüssels ein wie in einen unbalancierten Baum (siehe `treeInsert()` aus Übung 5). Damit ist Bedingung (1) erfüllt.
2. Wenn die Priorität des neuen Elements höher ist als die seines Vaterknotens: Führe eine Rotation aus, die das neue Element eine Ebene nach oben bewegt. Wenn das neue Element das linke Kind ist, muss eine Rechtsrotation ausgeführt werden und umgekehrt.
3. Wiederhole Schritt 2 bis entweder Bedingung (2) erfüllt oder das neue Element zur Wurzel des Treaps geworden ist. Da Bedingung (1) invariant gegenüber Rotationen ist, entsteht dadurch immer ein gültiger Treap.

Für die Festlegung der Prioritäten gibt es mehrere Möglichkeiten:

(A) Wenn man konstante Prioritäten verwendet, entsteht ein unbalancierter Baum.

(B) Wenn man Zufallszahlen verwendet, entsteht ein näherungsweise balancierter Baum.

(C) Wenn man die Häufigkeiten der Schlüssel verwendet, entsteht ein näherungsweise zugriffsoptimaler Baum.

(D) Wenn man bei jedem Zugriff auf ein Element dessen Priorität inkrementiert (und den Baum umstrukturiert, falls Bedingung (2) nicht mehr erfüllt ist), passt sich der Baum automatisch an variierende Zugriffsmuster an, indem häufig benutzte Schlüssel nach oben wandern.

Lösen Sie folgende Aufgaben und geben Sie die Implementation als File "`treap.py`" ab:

a) Implementieren Sie die aus der Vorlesung bekannten Operationen

```
newroot = treeRotateLeft(rootnode)
newroot = treeRotateRight(rootnode)
```

b) Implementieren Sie den obigen Einfügealgorithmus in Funktionen

```
newroot = randomTreapInsert(rootnode, key)
newroot = dynamicTreapInsert(rootnode, key)
```

die die Prioritäten nach Variante (B) bzw. (D) festlegen. Die `Node`-Klasse aus Übung 5 muss dazu um ein Attribut `priority` erweitert werden. Ist `key` bereits im Treap vorhanden, passiert bei der ersten Funktion nichts, während bei der zweiten die Priorität um Eins erhöht und der Baum gegebenenfalls umstrukturiert wird.

c) Erstellen Sie einen Baum, der alle Wörter des Files "`die-drei-musketiere.txt`" enthält. Das File wird folgendermaßen eingelesen und vorverarbeitet:

```
>>> s = open('die-drei-musketiere.txt').read()  # File einlesen
```

```
>>> for k in ',;.:-"\'!?':
...     s = s.replace(k, '')       # Sonderzeichen entfernen
>>> s = s.lower()                  # alles klein schreiben
>>> text = s.split()               # string in array von Wörtern umwandeln
```

Die Wörter in `text` werden nun in die beiden Treaps eingefügt:

```
>>> randomTreap = None
>>> dynamicTreap = None
>>> for word in text:
...     randomTreap  = randomTreapInsert(randomTreap, word)
...     dynamicTreap = dynamicTreapInsert(dynamicTreap, word)
```

Überprüfen Sie, dass beide Treaps die gleichen Elemente in korrekter Sortierung enthalten (benutzen Sie dazu in-order-Traversal, wie in der Vorlesung erläutert). Geben Sie die ersten 7 Ebenen von `dynamicTreap` mit den dazugehörigen Prioritäten (= Worthäufigkeiten) aus.

d) Welche Tiefe hätte ein perfekt balancierter Baum mit den gleichen Elementen? Vergleichen Sie dies mit der Tiefe der beiden Treaps. Berechnen Sie für beide Treaps die mittlere Zugriffszeit

$$\bar{t} = \frac{\sum_{w \,\epsilon\, \text{words}} h_w \, d_w}{\sum_{w \,\epsilon\, \text{words}} h_w}$$

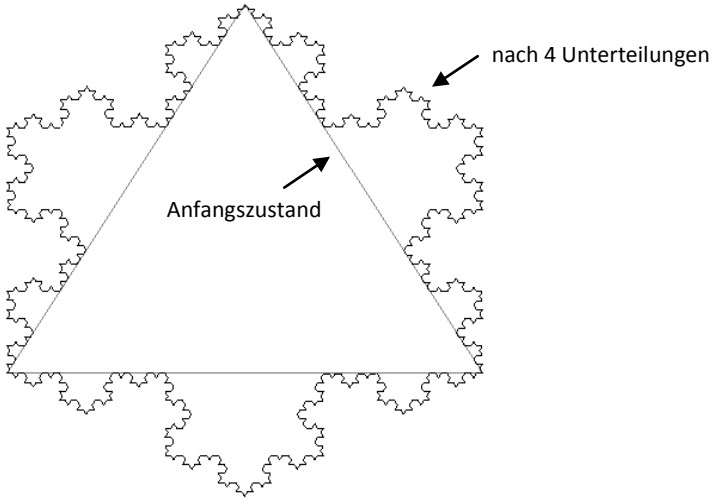
wobei h_w die Häufigkeit von Wort w und d_w seine Tiefe im Baum ist. Wird bestätigt, dass der dynamische Treap im Durchschnitt schneller ist?

Übung 7 Abgabe 29.5.2008

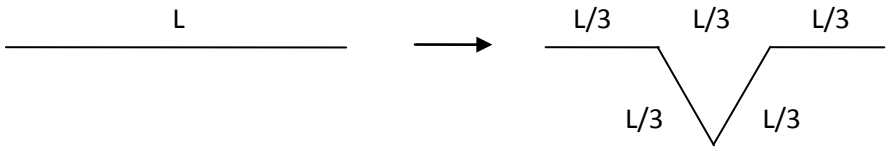
Aufgabe 1 – Rekursion und Iteration ? Punkte

a) Implementieren Sie die Fakultät mittels Rekursion (`recursiveFactorial(N)`) und mittels Iteration (`iterativeFactorial(N)`). Bestimmen Sie für beide Versionen die größte Zahl N, deren Fakultät berechnet werden kann, bzw. die größte Zahl N, deren Fakultät in weniger als etwa 10 Sekunden berechnet werden kann (falls ersteres zu lange dauert). Geben Sie Ihre Lösung in einem File "`factorial.py`" ab. Erklären Sie, warum beide Versionen sich unterschiedlich verhalten.

b) Entwickeln Sie ein Programm zur Berechnung der Koch-Schneeflocke:



Die Koch-Schneeflocke ist rekursiv definiert: Beginne mit dem gleichseitigen Dreieck $(0,0), (1,0), \left(\frac{1}{2}, \frac{\sqrt{3}}{2}\right)$. Teile jede vorhandene Strecke der Länge L in vier neue Strecken der Länge L/3 gemäß der Skizze (die Spitze bildet wieder ein gleichseitiges Dreieck):



Die entstehenden Strecken werden analog weiter unterteilt. Theoretisch kann dies unendlich oft wiederholt werden, praktisch wird man nach etwa 4 bis 8 Unterteilungen aufhören.

I. Geben Sie die formale Unterteilungsregel an: Wie werden aus gegebenen Endpunkten p_1, p_2 einer Strecke die neuen Punkte berechnet?

II. Implementieren Sie die Funktion `points=kochSnowflake(level)`. Die Anzahl der Unterteilungen wird durch den Parameter `level` bestimmt, die resultierende Punktliste als Python-Array zurückgegeben. Speichern Sie die Punktliste als Textfile (ein Punkt pro Zeile), zeichnen Sie die Schneeflocke mit Gnuplot (siehe Übung 2) oder einem anderen geeigneten Programm und exportieren sie das Bild in eine Bilddatei.[7] Geben Sie die Bilddatei und das File "`snowflake.py`" ab.

[7] Gnuplot-Befehle:

```
set term postscript color portrait # für Postscript-Ausgabe
set out "snowflake.eps"
set size square
set xrange [-0.2,1.2]
set yrange [-0.4:1.0]
plot "snowflake.txt" with lines
unset out
```

c) Merge Sort teilt die zu sortierenden Daten in zwei etwa gleich große Hälften, bearbeitet diese rekursiv und fügt das Ergebnis dann in linearer Zeit zusammen. Für die Laufzeit *T* ergibt sich daraus die Rekursionsformel

$$T(N) = 2T\left(\frac{N}{2}\right) + N$$

woraus, wie in der Vorlesung gezeigt, $T(N) = O(N \log N)$ für die Komplexität im schlechtesten Fall folgt. Nehmen wir an, ein modifizierter Algorithmus teilt die Daten im Verhältnis 1:3, seine Laufzeit ist also

$$T'(N) = T'\left(\frac{N}{4}\right) + T'\left(\frac{3N}{4}\right) + N$$

Was gilt jetzt für die Komplexität im schlechtesten Fall? Wie sieht es aus wenn

$$T''(N) = 4T''\left(\frac{N}{2}\right) + \frac{N}{2}$$

ist?

d) Wir haben in der Vorlesung einen rekursiven Algorithmus für die in-order-Traversierung eines Binärbaumes behandelt. Außerdem haben wir das Konzept der Iteratoren kennengelernt.[8]

I. Benutzen Sie einen Stack, um den rekursiven in einen iterativen Algorithmus zu transformieren, und geben Sie eine Python-Implementation an.

II. Implementieren Sie mit Hilfe der Stack-basierten Vorgehensweise einen Iterator für die in-order-Traversierung, den man wie folgt benutzen kann:

```
rootnode = None
... # insert data into the tree
for key in InOrderIterator(rootnode):
    print key
```

III. Schreiben Sie einen Unit Test für diesen Iterator, der die korrekte Reihenfolge der Schlüssel prüft.

IV. Wie muss man den Iterator verändern, um pre-order- bzw. post-order-Traversal zu erreichen?

Ihre Lösung soll im File "`in-order-traversal.py`" abgegeben werden.

[8] Die Implementation von Python-Iteratoren wird in Abschnitt 9.9 des Python-Tutoriums erläutert.

Übung 8 Abgabe 5.6.2008

Aufgabe 1 – Generizität ? Punkte

a) Wie kann man einen Stack bzw. eine Queue mit Hilfe einer Prioritätswarteschlange implementieren? Wie ein Array (d.h. Indexzugriff) mit Hilfe einer verketteten Liste? Skizzieren Sie entsprechende Adapter und erläutern Sie, welche Nachteile bezüglich Laufzeit/Komplexität man jeweils in Kauf nehmen muss.

b) Angenommen, ein Datentyp unterstützt den Vergleichsoperator "<". Wie kann man die übrigen Vergleichsoperatoren ("==", "!=", "<=", ">", ">=") nur mit Hilfe von "<" und Booleschen Verknüpfungen (`and`, `or`, `not`) ausdrücken? Was folgt daraus für die Gestaltung eines "minimal required interface"?

c) Die `sort()`-Funktion des Python-Arrays sortiert standardmäßig aufsteigend. Wenn eine andere Sortierung benötigt wird, kann man der Funktion einen Funktor übergeben, der die gewünschte Ordnung repräsentiert:

```
>>> array.sort(myOrdering)
```

Das required interface dieses Funktors ist ein sogenannter drei-wertiger Vergleich

$$\text{three\_way\_compare}(x,y) = \begin{cases} -1 & \text{wenn } x < y \\ 0 & \text{wenn } x = y \\ 1 & \text{wenn } x > y \end{cases}$$

Wenn kein Funktor angegeben ist, verwendet `sort()` die Python-Standardfunktion `cmp`. Absteigende Sortierung kann beispielsweise durch

```
>>> def greater(x,y): return -cmp(x,y)
>>> array.sort(greater)
```

erreicht werden. Implementieren Sie Funktoren für folgende Sortierungen sowie entsprechende Unit Tests und geben Sie Ihre Lösung im File "`sort-functor.py`" ab:

I. Gegeben ist ein Array mit positiven und negativen Zahlen. Es soll nach aufsteigendem Absolutbetrag sortiert werden.

II. Gegeben ist ein Array mit geraden und ungeraden Zahlen. Es soll so sortiert werden, dass zuerst alle geraden Zahlen aufsteigend und danach alle ungeraden Zahlen absteigend erscheinen (z.B. [1, 4, 3, 5, 7, 2, 6] → [2, 4, 6, 7, 5, 3, 1]).

Bonusaufgabe: Die Frösche im folgenden Bild sollen ihre Reihenfolge umkehren. In jedem Zug darf einer der Frösche ein oder zwei Felder nach links oder rechts springen, sofern das Zielfeld frei ist (bei einem Sprung über zwei Felder wird ein Frosch übersprungen). Wie schafft man dies in möglichst wenigen Zügen? Wie funktioniert dieses Prinzip bei einer beliebigen Anzahl von Fröschen? Sehen Sie einen Zusammenhang mit obiger Sortieraufgabe?

d) etwas über Iteratoren, evtl. mit map, reduce, oder filter???

e) Customized number class: dimension checker or automatic error propagation

Übung 9 Abgabe 12.6.2008

Aufgabe 1 – Graphen ? Punkte

a) Schreiben Sie die Adjazenzlisten für folgenden Graphen auf: ??? Skizzieren Sie den Graphen, der durch folgende Adjazenzlisten beschrieben ist (Haus vom Nikolaus???):

b) Ein Graph heißt *planar,* wenn er ohne Überschneidungen in der Ebene gezeichnet werden kann. Beweisen Sie, dass für jeden planaren Graphen gilt: $e \leq 3n - 6$, wo e die Anzahl der Kanten und n die Anzahl der Knoten bezeichnet. Benutzen Sie dafür die Eulersche Formel $n - e + f = 2$ (f ist die Anzahl der Flächen im Graphen, einschließlich der Außenfläche), die für jeden zusammenhängenden planaren Graphen gilt.

c) Angenommen ein Graphenalgorithmus hat die Komplexität $O(n^2 + n\, e \log e)$. Wie kann man das Resultat von b) benutzen, um diese Formel für planare Graphen zu vereinfachen?

d)

Übung 10 Abgabe 19.6.2008

Aufgabe 1 – ere ? Punkte

as

Übung 11 Abgabe 26.6.2008

Aufgabe 1 – erer ? Punkte

as

Übung 12 Abgabe 3.7.2008

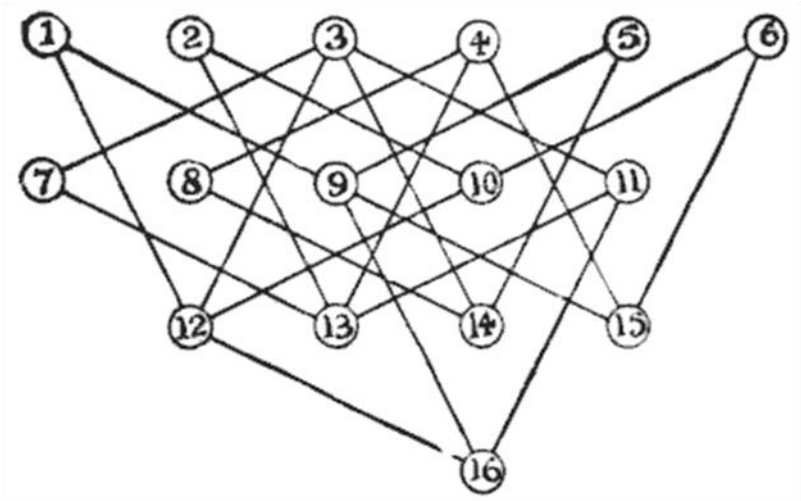
Aufgabe 1 – erer ? Punkte

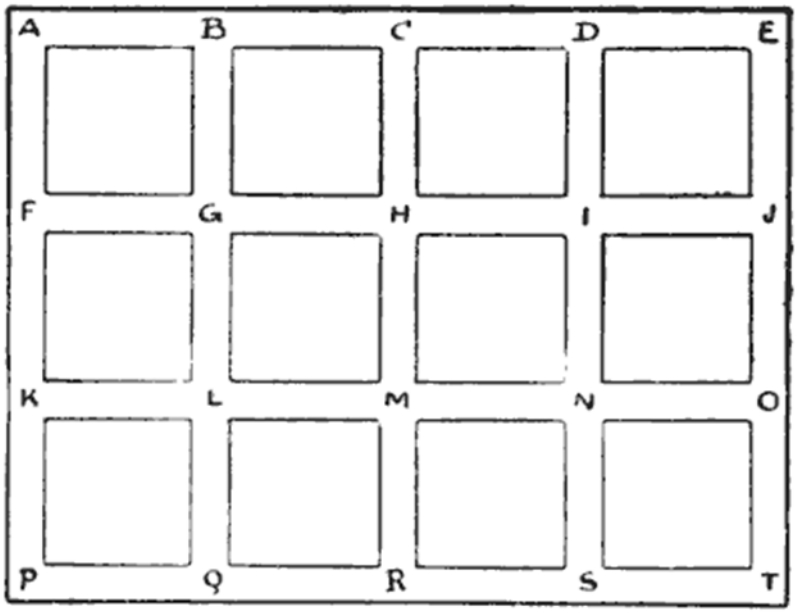
as

Übung 13 Abgabe 10.7.2008

Aufgabe 1 – erer ? Punkte

as





Übung 14 Abgabe 17.7.2008

Aufgabe 1 – erer ? Punkte

as

Lösungen