Übung 6 Abgabe 5.6.2014

Aufgabe 1 – Treaps 26 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 (Suchbaumbedingung).

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

Die Erfinder der Treap-Datenstruktur haben gezeigt, dass beide Eigenschaften gleichzeitig erfüllbar sind. Der grundlegende Einfügealgorithmus ist eine Kombination aus `treeInsert()` und `upHeap()`:

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 auf der nächsthöheren Ebene 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 *2 Punkte*

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

b) Implementieren Sie den obigen Einfügealgorithmus in Funktionen *10 Punkte*

```
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. Implementieren Sie geeignete Unit Tests.

c) Erstellen Sie Treaps, die alle Wörter des Files http://hci.iwr.uni-heidelberg.de/Staff/u-koethe/download/die-drei-musketiere.txt enthalten. Das File wird folgendermaßen eingelesen und vorverarbeitet: *4 Punkte*

```
# File einlesen und nach Unicode konvertieren (damit Umlaute korrekt sind)
>>> s = open('die-drei-musketiere.txt').read().decode('latin_1')
>>> 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)
```

Implementieren Sie eine Funktion `compareTreaps(treap1, treap2)`, die `True` zurückgibt, wenn beide Bäume die gleichen Elemente in gleicher Sortierung enthalten (verwenden Sie hierfür den Algorithmus `treeSort()` aus der Vorlesung). Testen Sie damit die beiden Treaps.

d) Welche Tiefe hätte ein perfekt balancierter Baum mit gleich vielen Elementen? Vergleichen Sie dies mit der Tiefe der beiden Treaps. Berechnen Sie für beide Treaps die mittlere Zugriffszeit *6 Punkte*

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

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

e) Implementieren Sie eine Funktion `treapTop(treap, my_priority)`, die ein Array zurückgibt, das alle `(key, priority)`-Paare aus dem Treap enthält, bei denen die Priorität (=Häufigkeit) mindestens `my_priority` ist. Wenden Sie diese Funktion mit `my_priority=500` auf die Datenstruktur `dynamicTreap` an. Sie werden erkennen, dass sich unter den häufigsten Wörtern viele nichtssagende Wörter wie "der", "und", "zu" befinden. In der Suchliteratur werden solche Wörter als *stop words* bezeichnet und bei der Volltextsuche ignoriert (siehe zum Beispiel http://de.webpageanalyse.com/blog/stoppwortlisten-in-9-sprachen). *4 Punkte*

Die Datei http://hci.iwr.uni-heidelberg.de/Staff/ukoethe/download/stopwords.txt enthält eine vordefinierte Liste von stop words, die Sie mittels

```
>>> stopWords = set(open('stopwords.txt').read().decode('latin_1').split())
```

einlesen können. Erstellen Sie einen weiteren dynamischen Treap `cleanedTreap`, wobei die stop words beim Einfügen übersprungen werden sollen. Benutzen Sie wiederum die Funktion `treapTop`, um die Wörter auszugeben, die mindestens 100 Mal vorkommen. Kann man jetzt den Charakter des Buches anhand der häufigsten Wörter einschätzen?

Aufgabe 2 – BucketSort — 16 Punkte

Gegeben sei eine Liste von Punkten, die im Einheitskreis gleichverteilt sind, d.h. jeder Punkt im Einheitskreis hat die gleiche Chance, in der Liste enthalten zu sein. Ihre Aufgabe besteht darin, die Punkte mittels BucketSort nach ihrem Abstand vom Koordinatenursprung $r = \sqrt{x^2 + y^2}$ (aufsteigend) zu sortieren. Dafür müssen Sie zunächst mit Hilfe des Python-Moduls `random` und dem sogenannten *rejection sampling* Testdaten erzeugen:

```
def createData(size):
    a = []
    while len(a) < size:
        x, y = random.uniform(-1, 1), random.uniform(-1, 1)
        r = sqrt(x**2 + y**2)
        if r < 1.0:
            a.append(r)
    return a
```

Die Funktion arbeitet folgendermaßen: Zunächst werden *x* und *y* als gleichverteilte Zufallszahlen im Intervall [-1, 1] gezogen. Damit sind die Punkte (*x,y*) gleichverteilt im Quadrat [-1,1]x[-1,1]. Durch den Test *r* < 1.0 wird geprüft, ob der Punkt (*x,y*) sogar im Einheitskreis (der ja eine Teilmenge des Quadrats ist) liegt. Nur dann wird er in die Liste *a* übernommen, andernfalls wird er ignoriert („rejection“). Der Einfachheit halber speichern wir nur die Abstände *r* in der Liste, weil die Koordinaten für die Aufgabe nicht mehr benötigt werden. Beachten Sie, dass die *r*-Werte im Intervall [0,1) nicht gleichverteilt sind.

a) Um BucketSort zu implementieren, brauchen Sie neben dem eigentlichen Algorithmus eine geeignete Funktion `bucketMap`, die die Sortierschlüssel auf Indizes im Bereich [0, *M*) abbildet, wenn *M* Buckets verwendet werden sollen. Da *r* im Intervall [0,1) liegt, bietet sich als naive Implementierung `index = int(r*M)` an. Allerdings werden die Schlüssel dann nicht gleichmäßig auf die Buckets verteilt – kleine Indizes kommen viel seltener vor als große (probieren Sie das aus!). Finden Sie eine bessere Formel, und begründen Sie, warum diese Formel im Mittel zu einer Gleichverteilung der Indizes führt. *6 Punkte*

b) Um experimentell zu prüfen, ob eine gegebene Funktion `bucketMap` zu einer gleichmäßigen Verteilung der Schlüssel auf die Buckets führt, kann man den χ^2-Test (sprich „Chi-Quadrat-Test“) verwenden. Angenommen, es sollen *N* Schlüssel auf *M* Buckets verteilt werden. Erfolgt die Aufteilung wirklich gleichmäßig, sollte jeder Bucket im Durchschnitt $c = N/M$ Schlüssel enthalten. Da die Schlüssel jedoch Zufallszahlen sind, wird das nur selten ganz exakt stimmen, und der χ^2-Test prüft, ob die Schwankungen innerhalb der erlaubten Grenzen bleiben. Man berechnet dazu die gewichtete Summe der quadratischen Abweichungen vom erwarteten Wert *c*, also die Größe *5 Punkte*

$$\chi^2 = \sum_{k=0}^{M-1} \frac{(n_k - c)^2}{c}$$

wobei n_k die tatsächliche Anzahl der Schlüssel im Bucket *k* sind. Wenn die Hypothese „Schwankungen liegen in den erwarteten Grenzen“ zutrifft, folgt χ^2 einer sogenannten Chi-Quadrat-Verteilung mit (*M*-1) Freiheitsgraden. Für genügend große *N* und *M* kann man dies weiter vereinfachen, weil dann die Größe $\tau = \sqrt{2\chi^2} - \sqrt{2M-3}$ näherungsweise wie eine Gaußsche Glockenkurve mit Mittelwert 0 und Standardabweichung 1 verteilt ist. Es gilt dann einfach: die Hypothese trifft mit 99.7%-iger Wahrscheinlichkeit *nicht* zu, die Daten sind also *nicht* gleichmäßig verteilt, wenn $|\tau| > 3$ ist.

Implementieren Sie diesen vereinfachten Test im File „`bucket_sort.py`“ mit einer Funktion `chiSquared(buckets)`, der das bereits gefüllte Bucketarray übergeben wird, und die `True` zurückgibt, wenn der Test bestanden wurde. Testen Sie für verschiedene Zufallsarrays und verschiedene *M*, ob Ihre Funktion `bucketMap` den Test besteht. Zeigen Sie außerdem, dass die naive Formel `int(r*M)` den Test nicht besteht. Erzeugen Sie die Zufallsarrays mit der oben angegebenen Funktion `createData`, die ebenfalls in `bucket_sort.py` enthalten sein soll.

c) Implementieren Sie `bucketSort` (siehe Vorlesung – vergessen Sie nicht, die Korrektheit Ihrer Implementation zu testen!) und zeigen Sie mit Hilfe des `timeit`-Moduls, dass die Laufzeit tatsächlich nur linear mit der Länge des Eingabearrays wächst, wenn $c = N/M$ genügend klein gewählt wird (Werte zwischen 2 und 6 sollten gute Ergebnisse liefern). Verwenden Sie sowohl die naive Formel `int(r*M)` als auch Ihre Funktion `bucketMap` und vergleichen Sie die Laufzeiten. Sie werden feststellen, dass die optimale Wahl der Funktion `bucketMap` nicht sehr kritisch ist – der χ^2-Test ist hier etwas zu pingelig. Die Lösung soll ebenfalls in `bucket_sort.py` enthalten sein. *5 Punkte*