Übung 6 — Abgabe 7.06.2019

Aufgabe 1 – Treaps — 28 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 (vergleiche `insert()` 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 Wahrscheinlichkeiten verwendet, mit der auf die Schlüssel zugegriffen wird, 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. Beim ersten Einfügen wird die Priorität mit 1 initialisiert.

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 = rotateLeft(rootnode)
newroot = rotateRight(rootnode)
```

b) Implementieren Sie die Klassen `RandomTreap` und `DynamicTreap`, die die Prioritäten nach Variante (B) bzw. (D) festlegen. Gehen Sie dabei von der Klasse `SearchTree` aus Übung 5 aus. Die `Node`-Klasse muss um ein Attribut `priority` erweitert werden, das `value`-Attribut kann dafür wegfallen, weil es in dieser Übung nicht benötigt wird. Die Funktion `insert()` realisiert die Treap-Semantik: Ist der `key` noch nicht im Treap enthalten, wird er nach der Suchbaumregel eingefügt und der Baum eventuell entsprechend der Prioritäten umstrukturiert. Ist der `key` bereits vorhanden, passiert beim `RandomTreap` nichts, während beim `DynamicTreap` die Priorität um Eins erhöht und der Baum gegebenenfalls umstrukturiert wird. Implementieren Sie außerdem geeignete Unit Tests. — *10 Punkte*

c) Auf Moodle finden Sie die Files `die-drei-musketiere.txt`, `casanova-erinnerungen-band-2.txt` und `helmholtz-naturwissenschaften.txt`. Wählen Sie eines dieser Files und erstellen Sie Treaps mit allen Wörtern des Textes. Ein Textfile wird folgendermaßen eingelesen und vorverarbeitet: *4 Punkte*

```
# File einlesen und nach Unicode konvertieren (damit Umlaute korrekt sind)
>>> filename = ...
>>> s = open(filename, encoding="latin-1").read()
>>> 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:

```
>>> rt = RandomTreap()
>>> dt = DynamicTreap()
>>> for word in text:
...     rt.insert(word)
...     dt.insert(word)
```

Implementieren Sie eine Funktion `compareTrees(tree1, tree2)`, die `True` zurückgibt, wenn beide Bäume die gleichen Elemente in gleicher Sortierung enthalten (gehen Sie hierfür vom Algorithmus `treeSort()` aus dem Skript aus). Vergleichen Sie damit die beiden Treaps.

d) Wie viele verschiedene Wörter enthält der Text? Welche Tiefe hätte ein perfekt balancierter Baum mit gleich vielen Elementen? Vergleichen Sie dies mit der Tiefe der beiden Treaps (dazu können Sie die Funktion `depth()` aus Übung 5 zu Ihren Klassen hinzufügen). Ermitteln Sie außerdem für beide Treaps die *mittlere* Tiefe *7 Punkte*

$$\bar{d} = \frac{\sum_{w \,\epsilon\, \text{words}} d_w}{n}$$

sowie die *mittlere* Zugriffszeit (=Tiefe gewichtet mit der Zugriffs-Häufigkeit)

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

wobei d_w die Tiefe von Wort w im Baum und h_w seine Häufigkeit (=Priorität im `DynamicTreap`) sowie n die Größe des Baums (=Anzahl der *verschiedenen* Wörter im Text) und N die Gesamtzahl der Wörter im Text sind. Wird bestätigt, dass der randomisierte Treap eine geringere mittlere Tiefe und der dynamische eine geringere mittlere Zugriffszeit hat?

e) Implementieren Sie in `DynamicTreap` eine Funktion `top(min_priority)`, die ein Array zurückgibt, das alle `(key, priority)`-Paare aus dem Treap enthält, bei denen die Priorität (=Häufigkeit) mindestens `min_priority` ist. Wenden Sie diese Funktion auf die Datenstruktur `dt` an. Wählen Sie `min_priority` so, dass ca. 100 Wörter ausgegeben werden. Sie werden unter den häufigsten Wörtern viele nichtssagende Wörter wie "der", "und", "zu" finden. In der Suchliteratur werden solche Wörter als *stop words* bezeichnet und bei der Volltextsuche ignoriert (siehe zum Beispiel de.wikipedia.org/wiki/Stoppwort). *5 Punkte*

Die Datei `stopwords.txt` auf Moodle enthält eine vordefinierte Liste von stop words, die Sie wie oben einlesen können. Erstellen Sie einen weiteren dynamischen Treap `cleanedTreap`, wobei die stop words beim Einfügen übersprungen werden (dazu empfiehlt es sich, die Liste der stop words zuerst in die Python-Datenstruktur `set` umzuwandeln). Benutzen Sie wiederum die Funktion `top()`, um die häufigsten Wörter auszugeben. Kann man jetzt den Charakter des Textes anhand der häufigsten Wörter einschätzen?