Humboldt-Universität zu Berlin
Institut für Informatik
Prof. Dr. Ulf Leser
E. Angriman, F. Nelles, M. Sänger, Dr. P. Schäfer

Berlin, den 19.06.2019

Übungen zur Vorlesung
Algorithmen und Datenstrukturen

Übungsblatt 6

Abgabe: Mittwoch, den 03.07.2019, bis 11:10 Uhr vor der Vorlesung im Hörsaal. Die Übungsblätter sind in Gruppen von 2/3 Personen zu bearbeiten. Die Lösungen sind auf nach Aufgaben getrennten Blättern abzugeben. Heften Sie bitte die zu einer Aufgabe gehörenden Blätter vor der Abgabe zusammen. Vermerken Sie auf allen Abgaben Ihre Namen, Ihre **CMS-Benutzernamen**, Ihre **Abgabegruppe** (z.B. AG123) aus Moodle, und den **Übungstermin** (z.B. Gruppe 2 oder Mo 13 Uhr bei M. Sänger), an dem Sie Ihre korrigierten Blätter zurückerhalten möchten.
Beachten Sie die Informationen auf der Übungswebseite (https://hu.berlin/algodat19).

Konventionen:

- Für ein Array A ist $|A|$ die Länge von A, also die Anzahl der Elemente in A. Die Indizierung aller Arrays auf diesem Blatt beginnt bei 1 (und endet also bei $|A|$).
- Mit der Aufforderung "Analysieren Sie die Laufzeit" ist gemeint, dass Sie eine möglichst gute obere Schranke der Zeitkomplexität angeben und diese begründen sollen.
- Der symmetrischen Vorgänger/Nachfolger ist der Knoten in einem binären Suchbaum mit dem nächstkleineren/nächstgrößeren Schlüssel.

Aufgabe 1 (Binäre Suchbäume) — 4 + 4 + 3 + 3 = 14 Punkte

In der Vorlesung haben Sie binäre Suchbäume kennengelernt, die ganze Zahlen als Schlüssel in den Konten speichern und die Operationen Einfügen, Suchen und Löschen von Schlüsseln unterstützen. Wir betrachten hier nur Suchbäume, die keine Duplikate enthalten, d.h., alle Schlüssel in den Knoten des Baums sind paarweise verschieden.

1. Zeigen Sie: Zu jedem binären Suchbaum T gibt es eine Reihenfolge seiner Schlüssel, so dass man durch Einfügen der Schlüssel in dieser Reihenfolge in einen anfangs leeren Baum den Baum T erhält.

2. Beweisen oder widerlegen Sie: Angenommen ein Knoten k eines binären Suchbaums hat einen symmetrischen Nachfolger, dann befindet sich der symmetrische Nachfolger immer im rechten Teilbaum von k.

3. Beweisen oder widerlegen Sie: Wenn man aus einem binären Suchbaum zwei beliebige Schlüssel x und y löscht, so erhält man unabhängig von der Löschreihenfolge stets den gleichen Suchbaum. Hierbei wird beim Löschen eines Knotens v mit zwei Kindern der Knoten v durch den symmetrischen Vorgänger ersetzt (vgl. Vorlesung).

4. Angenommen Sie suchen einen Schlüssel k in einem binären Suchbaum und finden ihn schließlich. Seien P die Schlüssel auf dem zugehörigen Suchpfad ab der Wurzel und L bzw. R die Schlüssel in allen Teilbäumen, die links bzw. rechts von diesem Suchpfad liegen (d.h. L, R und P sind paarweise disjunkt). Zeigen Sie, dass dann für $x \in L$, $y \in P$ und $z \in R$ nicht immer $x < y < z$ gelten muss, indem Sie ein möglichst kurzes Gegenbeispiel angeben.

Aufgabe 2 (AVL-Bäume) **7 + 7 = 14 Punkte**

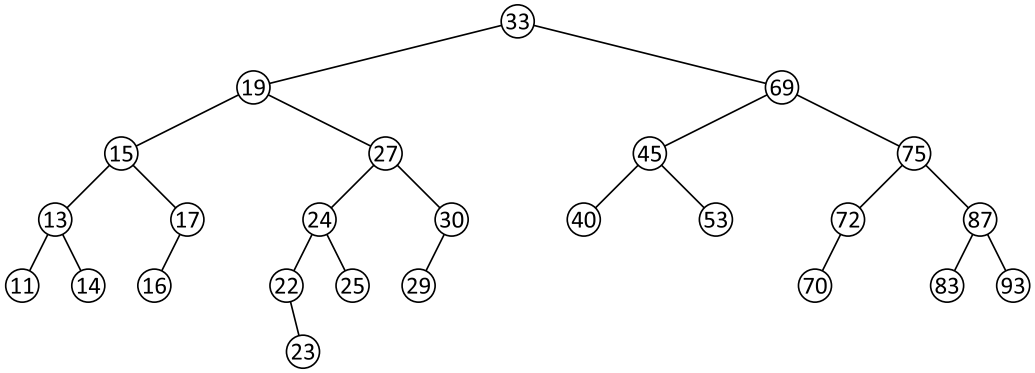
Führen Sie einen Schreibtischtest für die folgenden zwei Aufgaben zu AVL-Bäumen aus.

1. Sei T ein leerer AVL-Baum. Fügen Sie nacheinander die Schlüssel

   $$5, 17, 89, 23, 13, 2, 28, 26, 102, 18, 19, 117$$

   in T ein und zeichnen Sie den jeweiligen AVL-Baum nach jeder `insert`-Operation. Geben Sie zusätzlich alle Zwischenschritte in Form von Rotationen an. Annotieren Sie jeden Knoten mit seiner Balance (in der Vorlesung definiert als bal(p)).

2. Entfernen Sie nacheinander die Schlüssel $75, 87, 70$ aus dem unten gegebenen AVL-Baum. Zeichnen Sie den jeweiligen AVL-Baum nach jeder `delete`-Operation. Geben Sie zusätzlich alle Zwischenschritte in Form von Rotationen an. Annotieren Sie jeden Knoten mit seiner Balance. Falls ein Knoten mit zwei Kindern gelöscht wird, dann soll mit dem symmetrischen Vorgänger getauscht werden.



Aufgabe 3 (Implementierung Huffman-Kodierung) **4 + 4 + 4 = 12 Punkte**

In dieser Aufgabe sollen Sie mit Hilfe der Huffman-Kodierung einen Text kodieren und dekodieren. Die Huffmann-Kodierung ordnet einer festen Menge von Quellsymbolen jeweils Codewörter variabler Länge zu. Dazu werden zunächst die Häufigkeiten der einzelnen Symbole bestimmt und in einer Priority Queue verwaltet. Danach wird, wie in der Vorlesung kennengelernt (Folie 10, Priority Queues), in einem Bottom-Up Verfahren der binäre Huffman-Baum erstellt, um den Huffman Code für alle Symbole zu bestimmen.
Betrachten Sie für die Implementierung die auf der Übungswebsite und Moodle bereitgestellten Vorlagen `Huffman.java`. Die Klasse enthält die folgenden zwei privaten Variablen:

- Node *root*: Speichert die Wurzel des binären Huffman-Baums für die Kodierung.
- Map<Character, String> *code*: Speichert die Zuordnung eines Buchstaben zu seiner Kodierung.

Zusätzlich ist die Methode *encode* schon implementiert, in welcher die Häufigkeiten aller Buchstaben bestimmt werden und für jeden Buchstaben ein Knoten erstellt wird, welche nach Häufigkeit geordnet in einer Priority Queue verwaltet werden. Anschließend soll mit Hilfe der Methode *createTree* der binäre Huffman-Baum erstellt und danach mit Hilfe der Methode *createCode* für jeden Buchstaben die dazugehörige Kodierung bestimmt werden, um letztlich den übergebenen String zu kodieren.

Ergänzen Sie die Vorlage `Huffman.java` indem Sie folgende Funktionen implementieren:

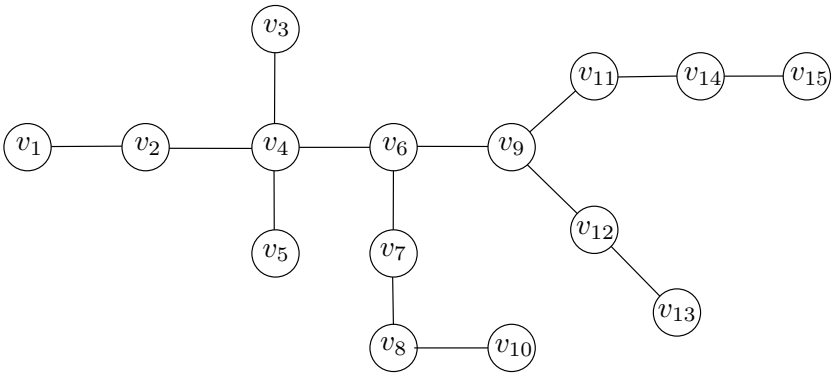
- void *createTree(*PriorityQueue<Entry> *pq)*: Erstellt den binären Baum für die Kodierung anhand der übergebenen Priority Queue und speichert die Wurzel in der privaten Variable *root*.
- void *createCode(*Node *node*, String *prefix)*: Speichert die Zuordung eines jeden Buchstaben zu seiner Kodierung in der privaten Variable *code*. Dazu wird der binäre Baum mit der aktuellen (Teil-)Kodierung traversiert. Der erstmalige Aufruf erfolgt mit der Wurzel des binären Huffman-Baums und einem leeren String.
- String *decode(*String *input)*: Dekodiert den String *input* (bestehend aus 0 und 1) anhand des binären Huffman-Baums.

Zum Testen können Sie die main-Methode in der Vorlage `Huffman.java` verwenden. Benutzen Sie dafür die Testdateien `lorem1.txt`, `lorem2.txt` und `lorem3.txt`. Sie können beliebige neue Variablen und Hilfsmethoden zur Klasse hinzufügen, dürfen jedoch keine außer den von Java bereitgestellten Standard-Bibliotheken verwenden.

Hinweis: Ihr Java-Programm muss auf dem Rechner *gruenau2* laufen. Kommentieren Sie Ihren Code, sodass die einzelnen Schritte nachvollziehbar sind. Die Abgabe des von Ihnen modifizierten Quellcodes `Huffman.java` erfolgt über Moodle.

Aufgabe 4 (Längster Pfad eines Baums) **7 + 3 = 10 Punkte**

Entwerfen Sie einen Algorithmus, der für einen beliebigen Baum mit n Knoten die Endpunkte eines längsten Pfades in diesem Baum berechnet. Für diese Aufgabe sei ein Baum ein ungerichteter, kreisfreier und zusammenhängender Graph. Beachten Sie, dass es in so einem ungerichteten Baum im Allgemeinen kein ausgezeichnetes Wurzelelement gibt. Die Länge eines Pfades ist die Anzahl der Kanten im Pfad. Betrachten Sie das dazu nachfolgende Beispiel:



In diesem Baum gibt es genau vier längste Pfade der Länge 7:

- $P_1 = v_1, v_2, v_4, v_6, v_9, v_{11}, v_{14}, v_{15}$,
- $P_2 = v_{15}, v_{14}, v_{11}, v_9, v_6, v_4, v_2, v_1$,
- $P_3 = v_{10}, v_8, v_7, v_6, v_9, v_{11}, v_{14}, v_{15}$ und
- $P_4 = v_{15}, v_{14}, v_{11}, v_9, v_6, v_7, v_8, v_{10}$.

Ihr Algorithmus sollte in diesem Beispiel also eine beliebige der folgenden Ausgaben liefern: „v_1, v_{15}“, „v_{15}, v_1“, „v_{10}, v_{15}“ oder „v_{15}, v_{10}“.

1. Beschreiben Sie einen Algorithmus (inklusive geeigneter Datenstrukturen der Eingabedaten), der das beschriebene Problem für n Knoten in Laufzeit $\mathcal{O}(n)$ löst. Begründen Sie, warum Ihr Algorithmus diese obere Schranke für die Laufzeit einhält.

2. Begründen oder widerlegen Sie, dass Ihr Algorithmus auch für beliebige ungerichtete, zusammenhängende (aber nicht notwendigerweise kreisfreie) Graphen stets die Endpunkte eines längsten Pfades ausgibt.