Humboldt-Universität zu Berlin
Institut für Informatik
Prof. Dr. Ulf Leser
M. Bux, B. Grußien, L. Heimberg, P. Schäfer

Berlin, den 29.06.2017

Übungen zur Vorlesung
Algorithmen und Datenstrukturen

Übungsblatt 6

Abgabe: Montag den 17.07.2017 bis 11:10 Uhr vor der Vorlesung im Hörsaal oder bis 10:45 Uhr im Briefkasten neben Raum 3.321, RUD25.

Die Übungsblätter sind in Gruppen von zwei (in Ausnahmen drei) Personen zu bearbeiten. Jedes Übungsblatt muss bearbeitet werden. (Sie müssen mindestens ein Blatt für wenigstens eine Aufgabe jedes Übungsblattes abgeben.) 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 die Namen und **CMS-Benutzernamen** aller Gruppenmitglieder, Ihre Abgabegruppe (z.B. AG123) aus Moodle, und den Übungstermin (z.B. Di 13 Uhr bei Marc Bux), zu dem Sie Ihre korrigierten Blätter zurückerhalten werden.

Beachten Sie die Informationen auf der Übungswebseite (https://hu.berlin/algodat17).

Aufgabe 1 (Binäre Suchbäume) $3 + 3 + 2 + 3 + 2 = 13$ Punkte

In dieser Aufgabe sollen Sie binäre Suchbäume in Java implementieren. Die dafür angelegte Klasse BinTree verwendet generische Typen, die das Comparable-Interface implementieren (z.B. Integer oder String). Weiterhin enthält sie einen Zeiger auf den Wurzelknoten root.
Jeder Knoten Node enthält das zu speichernde Element element sowie Zeiger parent, left und right auf den Elternknoten, den linken Kindknoten und den rechten Kindknoten. Falls ein Knoten keine Eltern oder keine Kinder hat, so zeigen die entsprechenden Zeiger auf null. Da es sich um einen binären Suchbaum handelt, gilt für jeden Knoten, dass das in ihm gespeicherte Element größer (kleiner) als alle Elemente im linken (rechten) Teilbaum ist.
Implementieren Sie die nachfolgenden Funktionen. Ergänzen Sie dazu den fehlenden Code in der Vorlage `BinTree.java`, welche Sie auf der Übungswebseite vorfinden. Sie können beliebige neue Variablen und Hilfsmethoden zur Klasse hinzufügen. Sie dürfen jedoch keine externen Bibliotheken sowie Klassen, die bereits einen Suchbaum implementieren, verwenden.

- contains(element): Überprüft, ob das Element element im binären Suchbaum enthalten ist. Gibt genau dann true zurück, wenn das Element gefunden wurde.
- insert(element): Fügt das Element element in den binären Suchbaum ein, falls es nicht bereits enthalten ist. Gibt genau dann true zurück, wenn das Element eingefügt wurde.
- successor(node): Gibt den symmetrischen Nachfolger des Knotens node zurück. Der symmetrische Nachfolger ist der Knoten mit dem kleinsten Element im rechten Teilbaum des Knotens node. Falls der rechte Teilbaum des Knotens node leer ist, so soll eine EmptyTree-Exception geworfen werden.
- delete(element): Löscht den Knoten mit dem Element element aus dem binären Suchbaum. Beim Löschen soll die Symmetrischer-Nachfolger-Methode angewendet werden. Die Methode soll genau dann true zurückgeben, wenn ein Knoten mit dem Element element entfernt wurde.

- inorder(lambda): Führt eine In-Order-Traversierung des binären Suchbaums durch und wendet dabei auf jedes Element den Lambda-Ausdruck lambda an. Bei einer In-Order-Traversierung wird beginnend bei der Wurzel zuerst der linke Teilbaum durchlaufen, dann der aktuelle Knoten besucht und schließlich der rechte Teilbaum durchlaufen. Auf diese Weise werden alle Knoten des Baums in einer eindeutigen Reihenfolge besucht. Die Methode wendet den Lambda-Ausdruck lambda in der Reihenfolge auf die Elemente des Baums an, in der ihre Knoten besucht werden. Die Ergebnisse werden als Liste zurückgegeben.

Stellen Sie sicher, dass alle Testfälle in der main-Methode der Datei BinTree.java erfolgreich durchlaufen. Achten Sie außerdem auf Randbedingungen und Spezialfälle, die von den Testfällen vielleicht nicht vollständig abgedeckt werden.
Hinweis zur Abgabe: 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 Source-Codes `BinTree.java` erfolgt über Moodle.

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

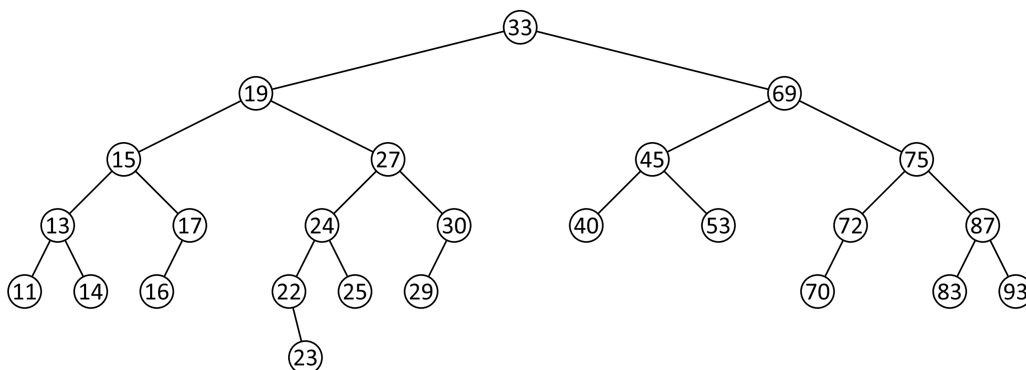
Führen Sie einen Schreibtischtest für die folgenden zwei Aufgaben zu AVL-Bäumen aus.

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

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

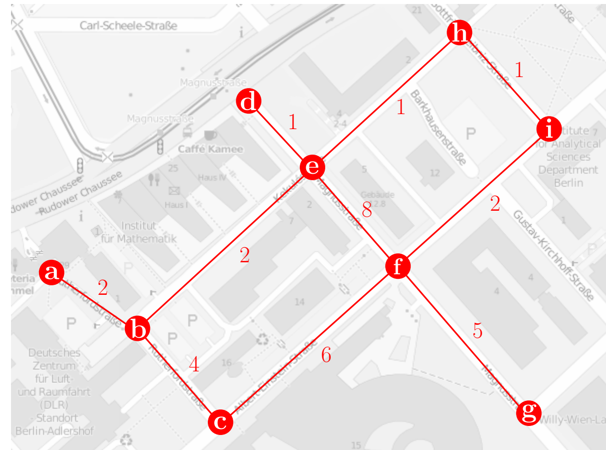
   in T ein und zeichnen Sie den jeweiligen AVL-Baum nach jeder `insert`-Operation.

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. Annotieren Sie jeden Knoten mit seiner Balance (in der Vorlesung definiert als bal(p)). Falls ein Knoten mit zwei Kindern gelöscht wird, dann soll mit dem symmetrischen Vorgänger getauscht werden.



Aufgabe 3 (Dijkstra-Algorithmus) **6 + 7 = 13 Punkte**

Stellen Sie sich vor, dass es in Ihrer Lieblingsstadt keine Verkehrszeichen gibt, d.h., an allen Kreuzungen muss die Regel rechts-vor-links beachtet werden. Außerdem nehmen wir an, dass an jeder Kreuzung maximal vier Straßen aufeinander treffen.
Wir modellieren die Straßen der Stadt wie folgt als ungerichteten Graphen: Jede Kreuzung wird ein Knoten des Graphen und jede Straße, die zwei Kreuzungen verbindet, wird eine Kante. Wir nehmen an, dass Straßen unterschiedlich aufwändig zu durchfahren sind. Dies modellieren wir durch nichtnegative Kantenkosten. Betrachten Sie das folgende Beispiel. Beachten Sie nur die roten Knoten und Kanten (z.B. hat Kreuzung i nur zwei Zufahrtsstrassen):



Nun führen wir noch Kosten für das Überfahren einer Kreuzung ein. Die Abbiegekosten von x über y nach z werden durch die Funktion $AK(x, y, z)$ berechnet, die Sie als gegeben annehmen können:

- Muss man auf niemanden achten, dann kostet das Überfahren der Kreuzung 1 – z.B. beim Rechtsabbiegen von b über e zu f oder beim Linksabbiegen von i über h nach e.
- Muss man auf eine Richtung achten, dann kostet das Überfahren der Kreuzung 2 – z.B. beim Geradeausfahren von b über e zu h oder beim Linksabbiegen von a über b zu e.
- Muss man auf zwei Richtungen achten, dann kostet das Überfahren der Kreuzung 3 – z.B. beim Linksabbiegen von e über f zu i.

Beispielsweise würde der Pfad $e \to f \to i$ im obigen Graph $8+3+2$ kosten, der Pfad $a \to b \to c$ kostet $2+1+4$, der Pfad $i \to h \to e$ kostet $1+1+1$.

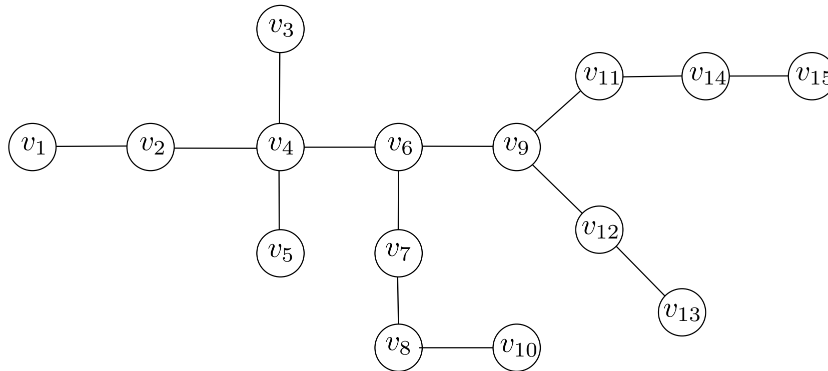
1. Angenommen, der Algorithmus von Dijkstra auf Foliensatz 13 (Priority Queues), Folie 27 wird wie folgt um die Abbiegekosten ergänzt: Für jeden Knoten k wird neben der Distanz auch sein Vorgängerknoten gespeichert. Sei y dieser Vorgängerknoten, also der Knoten, von dem aus zuletzt die Distanz zu k verringert wurde. Dann ergibt sich new_dist aus der Distanz zu k, den Kosten w der Kante zwischen k und f, sowie den Abbiegekosten von y über k nach f: $new_dist = A[k] + w + AK(y, k, f)$.

   Zeigen Sie, dass die beschriebene Modifikation des Dijkstra-Algorithmus *nicht* die korrekte Distanz (also die Länge eines kürzesten Pfads) von Startknoten x zu allen Knoten des Graphs berechnet.

2. Beschreiben Sie, wie ein gegebener Graph G modifiziert werden kann, um die Abbiegekosten direkt in die Kantenkosten zu integrieren, damit der unmodifizierte Dijkstra-Algorithmus die korrekten Distanzen für das Szenario mit Abbiegekosten berechnet.

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

In dieser Aufgabe sollen Sie einen Algorithmus entwerfen, der für einen beliebigen Baum mit n Knoten die Endpunkte eines längsten Pfades in diesem Baum berechnet. Ein Baum ist ein ungerichteter, kreisfreier und zusammenhängender Graph. Beachten Sie, dass es in einem Baum im Allgemeinen kein ausgezeichnetes Wurzelelement gibt. Die Länge eines Pfades ist die Anzahl der Kanten im Pfad. Betrachten Sie das 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 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 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 Graphen stets die Endpunkte eines längsten Pfades ausgibt.