Humboldt-Universität zu Berlin
Institut für Informatik
Prof. Dr. Marius Kloft
Bux, Grußien, Schäfer, Tschorsch, Völlinger

Berlin, den 17.05.2016

Übungen zur Vorlesung
Algorithmen und Datenstrukturen

Übungsblatt 3

Abgabe: Montag den 30.5.2016 bis 10:55 Uhr vor der Vorlesung im Hörsaal oder bis 10:45 Uhr im Briefkasten (RUD 25, Raum 3.321). Die Übungsblätter sind in Gruppen von zwei (in Ausnahmefällen auch drei) 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 Matrikelnummern, den Namen Ihrer Goya-Gruppe und an welchem Übungstermin (Wochentag, Uhrzeit, Dozent) Sie die korrigierten Abgaben abholen möchten. Beachten Sie auch die aktuellen Hinweise auf der Übungswebsite unter https://hu.berlin/algodat16.

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 $|A|$). Bitte beginnen Sie die Indizierung der Arrays in Ihren Lösungen auch bei 1.
- Mit der Aufforderung „analysieren Sie die Laufzeit“ ist gemeint, dass Sie eine möglichst gute obere Schranke der Zeitkomplexität angeben sollen und diese begründen sollen.

Aufgabe 1 (Skip-Listen) — (3 + 3 + 3 + 3) + (5 + 5) = 22 Punkte

In dieser Aufgabe sollen Sie eine *Skip-Liste* zum effizienten Speichern sortierter Integer-Werte implementieren. Diese zu implementierende Skip-Liste verwendet intern die Datenstruktur der *sortierten doppelt verketteten Liste*. Implementieren Sie zuerst in Teilaufgabe (a) eine solche sortierte doppelt verkettete Liste und verwenden Sie diese anschließend in Teilaufgabe (b) zur Implementierung der Skip-Liste.

(a) In dieser Teilaufgabe sollen Sie zunächst die Datenstruktur der sortierten doppelt verketteten Liste zum Speichern von Integer-Werten implementieren (vgl. Abbildung 1). Eine solche Liste lässt sich als eine Menge von Knoten repräsentieren, von denen jeder einen Integer-Wert und zwei Zeiger (auf den nächsten und auf den vorherigen Knoten) speichert. Für jeden Knoten ist der Wert des vorherigen/nächsten Knotens stets kleiner/größer als der Wert des Knotens. In dieser Implementierung soll die Liste also keine Duplikate (Knoten mit dem gleichen Werten) enthalten. Außerdem soll die Liste sowohl über einen ausgezeichneten Start-Knoten (FIRST) als auch einen ausgezeichneten End-Knoten (LAST) verfügen. Dies ist später für die Implementierung der Skip-Liste hilfreich. Auch eine leere Liste enhält diese beiden Knoten.

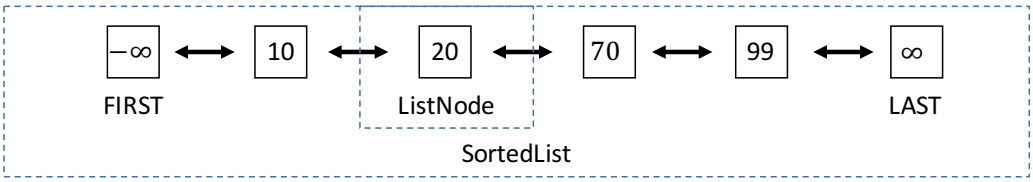

Abbildung 1: Eine sortierte doppelt verkettete Liste.

Implementieren Sie die nachfolgenden Funktionen für die Knoten einer sortierten doppelt verketteten Liste. Ergänzen Sie dazu den fehlenden Code in der Vorlage `ListNode.java`, welche Sie auf der Website vorfinden, unter Ausnutzung der existierenden Funktionen-Stubs. Beachten Sie, dass es sich bei den zu implementierenden Funktionen nicht um Funktionen der Liste, sondern um Funktionen eines Knotens in der Liste handelt. Der Grund dafür ist, dass diese Funktionen in Teilaufgabe (b) zur Implementierung der Skip-Liste benötigt werden.

- `search(value)`: Liefert ausgehend vom aktuellen Knoten den Knoten mit dem gesuchten Wert `value`, falls vorhanden, bzw. andernfalls den Knoten mit dem zu `value` nächstkleineren Wert. Beachten Sie, dass der gesuchte Knoten sowohl vor als auch hinter dem aktuellen Knoten liegen kann.
- `insertAfter(value)`: Fügt direkt hinter dem aktuellen Knoten einen neuen Knoten mit dem Wert `value` ein und gibt ihn zurück.
- `insert(value)`: Fügt einen neuen Knoten mit Wert `value` ausgehend vom aktuellen Knoten sortiert in die Liste ein und gibt ihn zurück. Falls der Wert `value` bereits in der Liste vorhanden ist, so wird keine Veränderung an der Liste vorgenommen und der existierende Knoten mit dem Wert `value` wird zurückgegeben.
- `delete(value)`: Entfernt den Knoten mit dem Wert `value`, falls vorhanden, ausgehend vom aktuellen Knoten.

Sie benötigen zum Kompilieren auch die Dateien `LinkedList.java` und `SortedList.java`. Zum Testen können Sie die `main`-Methode in der Vorlage `SortedList.java` verwenden.

Hinweis: Implementieren Sie zuerst die Methoden `search` und `insertAfter` und nutzen Sie diese für die Methoden `insert` und `delete`.

Hinweis zur Abgabe: Ihr Java-Programm muss unter *Java 1.7* auf dem Rechner *gruenau2* laufen. Kommentieren Sie Ihren Code, sodass die einzelnen Schritte nachvollziehbar sind. Die Abgabe der von Ihnen modifizierten Datei `ListNode.java` erfolgt über Goya.

(b) Die naive Implementierung der in Teilaufgabe (a) implementierten Funktion `search` benötigt $\mathcal{O}(n)$ Operationen. Auch im Average-Case lässt sich eine sortierte doppelt verkettete Liste nur in linearer Zeit durchsuchen. Aufbauend auf Ihrer Implementierung der sortierten doppelt verketteten Liste sollen Sie nun eine Skip-Liste implementieren. Eine Skip-Liste ist eine Datenstruktur, die im Average-Case in logarithmischer Zeit durchsuchbar ist.

Dafür enthält die Skip-Liste mehrere Ebenen. Auf der untersten Ebene kommen alle Knoten vor, während auf den oberen Ebenen immer weniger Knoten vorkommen. Kommt ein Knoten auf einer Ebene vor, so enthalten auch alle darunter liegenden Ebenen den Knoten. Jede Ebene wird durch eine sortierte doppelt verkettete Liste mit jeweils ausgezeichneten Start- und Endknoten realisiert. Zusätzlich enthält nun jeder Knoten einer Ebene sowohl einen Zeiger auf den Knoten desselben Wertes in der darunterliegenden Ebene als auch einen Zeiger auf den Knoten desselben Wertes in der darüberliegenden Ebene.

Eine Skip-Liste mit drei Ebenen wird exemplarisch in Abbildung 2 dargestellt. Hier kommt der Knoten mit dem Wert 20 beispielsweise auf allen drei Ebenen vor, während der Knoten mit dem Wert 70 nur auf der untersten Ebene enthalten ist.

Um nun den Wert 70 in dieser Skip-Liste zu finden, sucht man zunächst auf der obersten Ebene. Da die 70 auf dieser Ebene nicht gefunden werden kann, geht man ausgehend vom nächstkleineren Wert, der 20, eine Ebene tiefer. Diese Suche wiederholt man auf jeder Ebene bis man den Wert gefunden hat oder den Wert auch auf der untersten Ebene nicht finden konnte. Da die 70 auf der untersten Ebene gefunden wurde, wird diese zurückge-

geben. Im Average-Case reduziert sich durch dieses Vorgehen die Laufzeit der Methode `search` auf $\mathcal{O}(\log n)$.

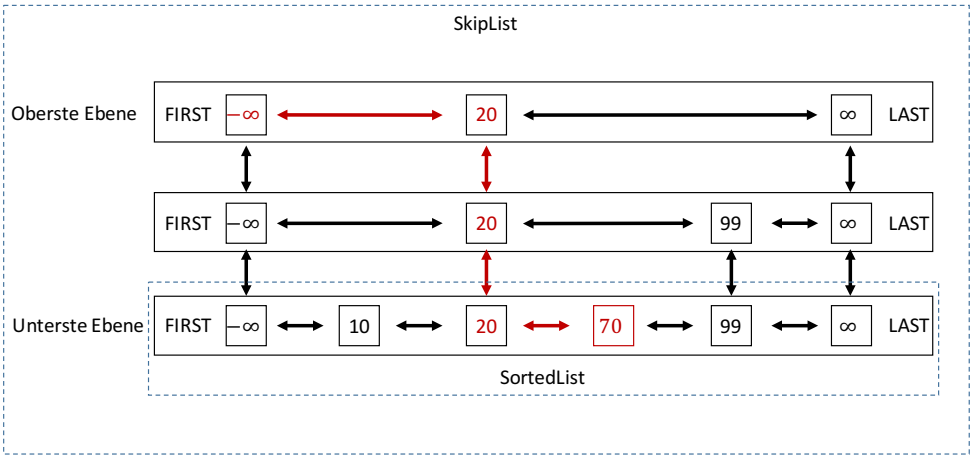

Abbildung 2: Eine Skip-Liste basiert auf sortierten doppelt verketteten Listen. Der Suchpfad zum Wert 70 ist rot unterlegt.

Um einen neuen Knoten mit neuem Wert einzufügen, müssen zuerst die beiden Knoten auf der untersten Ebene gefunden werden, zwischen denen der neue Knoten einzufügen ist. Beispielsweise wird der Knoten mit der 80 in die Skip-Liste in Abbildung 3 zwischen den Knoten mit der 70 und der 99 auf der untersten Ebene eingefügt. Anschließend muss entschieden werden, ob auch auf den darüberliegenden Ebenen ein neuer Knoten eingefügt werden soll. Das wird über einen Münzwurf entschieden. Dabei wird folgende Regel angewandt: Ein neuer Knoten wird immer auf der untersten Ebene eingefügt. Für jede darüberliegende Ebene, wird mit 50 % Wahrscheinlichkeit (Münzwurf) entschieden, ob ein neuer Knoten auch hier eingefügt werden soll. Somit landet ein Wert mit 100 % Wahrscheinlichkeit auf der ersten (untersten) Ebene, mit 50 % auf der zweiten Ebene, mit 25 % auf der dritten Ebene, mit 12,5 % auf der vierten Ebene, usw. Im Average Case reduziert sich durch dieses Vorgehen die Laufzeit der Operation `insert` auf $\mathcal{O}(\log n)$.

Implementieren Sie die folgenden Funktionen für eine Skip-Liste zum Speichern von Integer-Werten in Java:

- `search(value)`: Sucht ausgehend von der obersten Ebene nach dem Knoten mit dem Wert `value`. Falls vorhanden gibt die Funktion den Knoten mit dem Wert `value` auf der untersten Ebene zurück. Andernfalls liefert die Methode den Knoten mit dem zu `value` nächstkleineren Wert (ebenfalls auf der untersten Ebene). Ihre Implementierung soll im Average Case $\mathcal{O}(\log n)$ Operationen benötigen.
- `insert(value)`: Fügt einen neuen Knoten mit Wert `value` sortiert auf der untersten Ebene der Skip-Liste ein. Anschließend wird über die oben beschriebene Regel entschieden, in welchen höher gelegenen Ebenen ebenfalls neue Knoten mit dem Wert `value` eingefügt werden. Existieren hierfür nicht ausreichend viele höher gelegene Ebenen, so müssen diese neu erstellt werden. Der auf der untersten Ebene neu eingefügte Knoten mit dem Wert `value` wird zurückgeben. Falls der Wert `value` bereits im Voraus in der untersten Ebene vorhanden ist, so wird keine Veränderung an der Liste vorgenommen und der existierende Knoten mit dem Wert `value` wird zurückgegeben. Ihre Implementierung soll im Average Case $\mathcal{O}(\log n)$ Operationen benötigen.

Ergänzen Sie den fehlenden Code in der Vorlage `SkipList.java`, welche Sie auf der Website vorfinden, unter Ausnutzung der existierenden Funktionen-Stubs. Da jede Ebene der

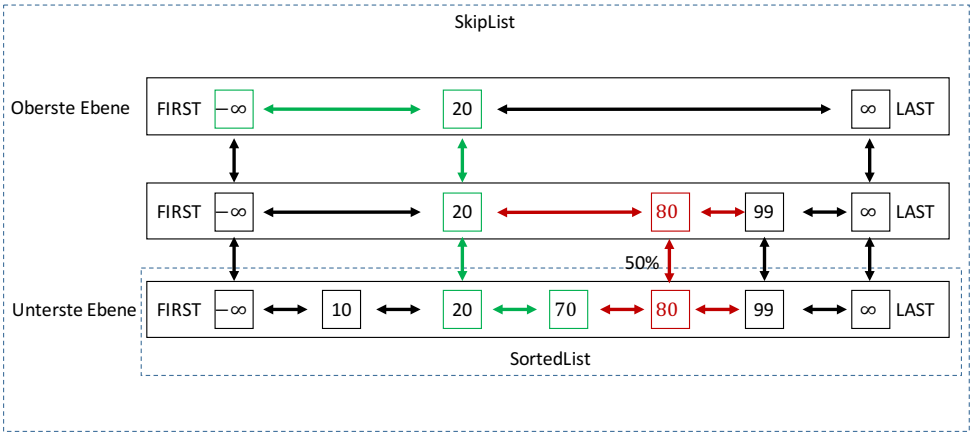

Abbildung 3: Eine Skip-Liste, in die der Wert 80 eingefügt wird. Der Suchpfad zur Stelle, an dem der Wert 80 eingefügt werden soll, ist grün unterlegt. Ein neuer Knoten mit dem Wert 80 wird auf der untersten Ebene nach dem Knoten mit dem Wert 70 eingefügt. Nach erfolgreichem Münzwurf wird auch auf einer darüberliegenden Ebene ein neuer Knoten mit dem Wert 80 eingefügt. Da der nächste Münzwurf fehlschlägt wird kein neuer Knoten auf der nächsthöheren Ebene eingefügt. Die geänderten Elemente (Knoten und Pointer) nach der Einfüge-Operation sind rot markiert.

Skip-Liste durch eine sortierte Liste repräsentiert wird, benötigen Sie zum Kompilieren auch die Dateien `LinkedList.java`, `SortedList.java` sowie die von Ihnen vervollständigte `ListNode.java` aus Teilaufgabe (a). Zum Testen können Sie die `main`-Methode in der Vorlage `SkipList.java` verwenden. Auch wenn Sie die Teilaufgabe (a) nicht komplett lösen konnten, können Sie Punkte für diese Teilaufgabe bekommen.

Hinweis: Um zu entscheiden, ob ein Knoten auf einer höheren Ebene eingefügt werden soll, können Sie die Methode `flipCoin` verwenden, die mit 50 % Wahrscheinlichkeit `true` oder `false` liefert. Nutzen Sie ebenso die Hilfsmethode `createNewLayer` um eine neue Ebene zur Skip-Liste hinzuzufügen.

Hinweis zur Abgabe: Ihr Java-Programm muss unter *Java 1.7* auf dem Rechner *gruenau2* laufen. Kommentieren Sie Ihren Code, sodass die einzelnen Schritte nachvollziehbar sind. Die Abgabe der von Ihnen modifizierten Datei `SkipList.java` erfolgt über Goya.

Aufgabe 2 (Listen) **8 + 2 = 10 Punkte**

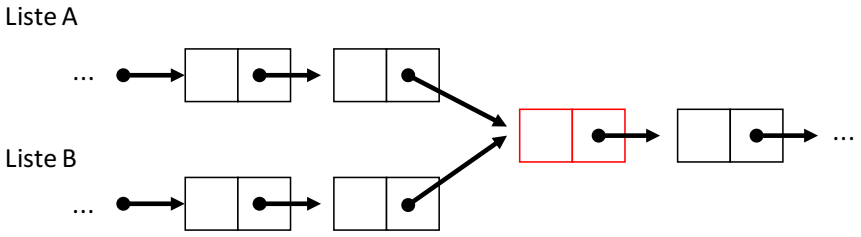

Abbildung 4: Zwei einfach verkettete Listen A und B. Es existiert ein Knoten (rot markiert), ab dem die Listen übereinstimmen.

Gegeben seien zwei einfach verkettete Listen A und B, die mindestens einen gemeinsamen Knoten enthalten (vgl. Abbildung 4).

(a) Erläutern Sie, wie Sie den ersten gemeinsamen Knoten finden können, ab dem die beiden einfach verketteten Listen übereinstimmen. Beachten Sie, dass die beiden Listen nicht die gleiche Länge haben müssen. (Es ist nicht notwendig und nicht ausreichend hier Pseudocode anzugeben.)

Hinweis: Die Punktevergabe erfolgt gestaffelt nach Effizienz bezüglich der Laufzeit ihrer Lösung.

(b) Analysieren Sie die Laufzeit Ihres vorgestellten Algorithmus.

Aufgabe 3 (Dynamische Programmierung) **10 Punkte**

Im Katz-und-Maus-Spiel sitzt eine Katze im unteren, linken Feld eines rechteckigen Spielbretts bestehend aus $n \cdot m$ Feldern mit $n, m \geq 2$. Ihre Beute sitzt, in Schockstarre und in die Enge getrieben, im rechten, oberen Feld. Die vom Hunger getriebene Katze kann sich stets nur in Richtung der Maus, also nach oben oder nach rechts bewegen. Da es kürzlich geregnet hat, befinden sich auf vereinzelten Feldern Wasserpfützen, wodurch diese Felder für die wasserscheue Katze unpassierbar sind (vgl. Abbildung 5).

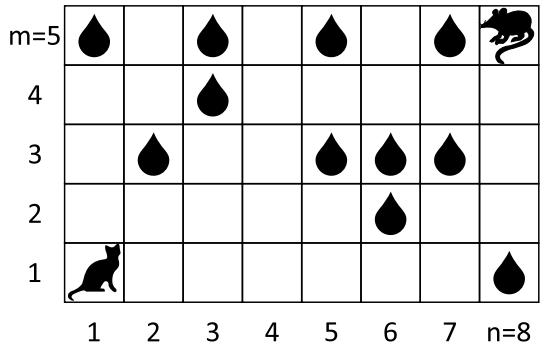

Abbildung 5: Darstellung einer Instanz des Katz-und-Maus-Spiels. Durch Wasserpfützen unpassierbar gewordene Felder werden durch einen Wassertropfen gekennzeichnet.

Entwerfen Sie einen Algorithmus, der einen von der Katze begehbaren Weg von der Katze zur Maus berechnet, falls es einen solchen Weg gibt. Gehen Sie dabei davon aus, dass das Spielfeld als zweidimensionales Array F von Boolean-Werten gegeben ist, in dem genau die unpassierbaren Felder, auf denen sich eine Wasserpfütze befindet, den Wert *true* innehaben. Die Katze befindet

sich zu Beginn auf Feld $F[1][1]$, während sich die Maus auf Feld $F[n][m]$ befindet. Sowohl die Wasserpfützen als auch die Maus ändern ihre jeweiligen Positionen während des Spiels nicht. Der zu ermittelnde Weg soll als Liste von Koordinaten der Form (x, y) zurückgegeben werden, bzw. `null`, falls kein Weg existiert. Geben Sie Ihren Algorithmus in Pseudocode an, beschreiben Sie dessen Funktionsweise und analysieren Sie seine Laufzeit.

Hinweis: Die Punktevergabe erfolgt gestaffelt nach Effizienz *bezüglich der Laufzeit*. Sie dürfen unbegrenzt zusätzlichen Speicherplatz verwenden.

Aufgabe 4 (Parkplatzsuche) **8 Punkte**

Stellen Sie sich folgendes Szenario vor: Sie wohnen an einer unendlich langen Straße und kommen nach einem anstrengenden Tag mit dem Auto zu Hause an. Leider ist weit und breit kein Parkplatz in Sicht. Aus Erfahrung wissen Sie jedoch, dass es irgendwo auf dieser Straße einen Parkplatz geben muss. Da Sie müde und zugleich umweltbewusst sind, wollen Sie diesen Parkplatz finden und dabei möglichst wenig Fahrtweg zurücklegen.

Wenn Sie wüssten, in welcher Richtung sich dieser Parkplatz befindet, würden Sie direkt in diese Richtung fahren und dabei an L belegten Parkplätzen vorbeifahren (vgl. Abbildung 6).

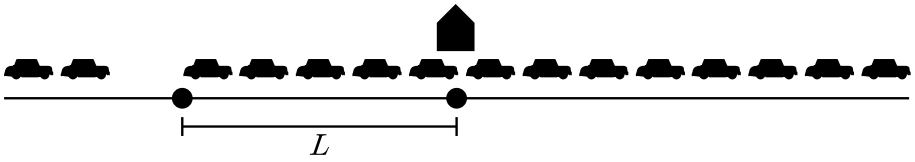

Abbildung 6: Darstellung der Ausgangslage bei der Parkplatzssuche.

Entwerfen Sie einen Algorithmus (hier genügt eine informelle Beschreibung), mit dem Sie höchstens einen Weg von $10 \cdot L$ zurücklegen müssen, um den Parkplatz zu finden. Beweisen Sie, dass Ihr Algorithmus die geforderte Eigenschaft besitzt.

Hinweis: Sie starten direkt vor Ihrem Haus. Die Straße hat keine Abzweigung und Sie können nur vor- oder zurückfahren. Beachten Sie außerdem, dass Sie L nicht kennen.