Prof. Dr. Gerd Stumme, Folke Eisterlehner, Dominik Benz
Fachgebiet Wissensverarbeitung

22.06.2009

11. Hausübung „Algorithmen und Datenstrukturen“

Sommersemester 2009

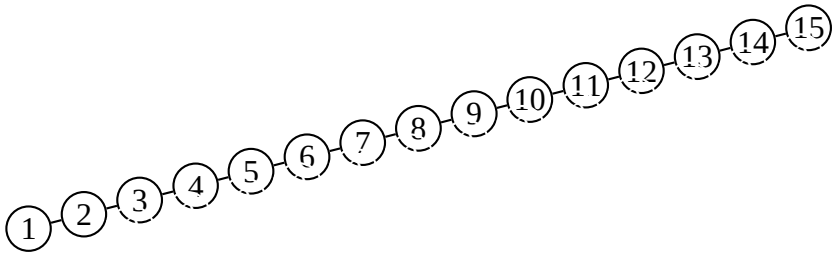
Abgabetermin: Montag, 29.06.2009, 10:00 Uhr

1 Suchbäume

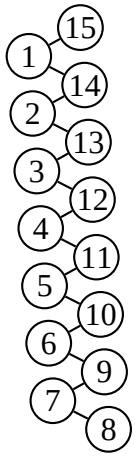
Aufgabe 1 (Einfügereihenfolge)

Betrachten Sie folgende Suchbäume und geben Sie eine mögliche Reihenfolge an, in der die Schlüssel $K = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15\}$ in den jeweiligen Baum eingefügt wurden. Geben Sie zudem an, ob es für jeden Baum *genau eine* oder verschiedene Eingabefolgen gibt, die zum jeweiligen Baum führen.

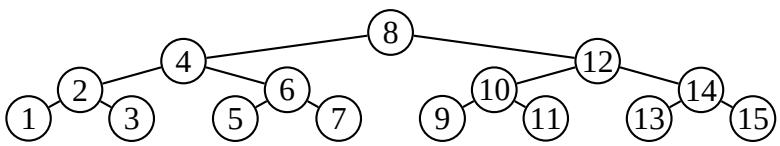
a)



b)



c)



Aufgabe 2 (Binärer Suchbaum)

Schreiben Sie eine generische Klasse **BinarySearchTree**, welche das Interface **SearchTree** von der Webseite zur Vorlesung implementiert.

Insbesondere sind also folgende Methoden zu implementieren:

boolean insertKey(K key): Falls der Schlüssel **key** noch nicht im Suchbaum enthalten ist, wird für diesen Schlüssel ein neuer Knoten im Suchbaum an passender Stelle eingefügt und **true** zurückgegeben.

Andernfalls wird der Suchbaum nicht verändert und **false** zurückgegeben.

boolean lookupKey(K key): Falls der Schlüssel **key** im Suchbaum enthalten ist, wird **true** zurückgegeben, andernfalls **false**.

In beiden Fällen wird der Suchbaum nicht verändert.

Die Knoten des Suchbaums sollen durch eine geschachtelte Klasse **BinarySearchTreeNode** modelliert werden, welche das Interface **BinaryTreeNode** von der Vorlesungsseite implementiert.

Im Gegensatz zur Vorlesung soll es *keine* speziellen „leeren“ Pseudoknoten (dort **head** und **nullNode** genannt) geben. Entsprechend werden leere Bäume durch **null**-Zeiger dargestellt.

Implementieren Sie zusätzlich folgende Methode:

boolean insertKey(K[] keys): Ruft die Methode **insertKey(k)** für jeden übergebenen Schlüssel **k** $\in$ **keys** auf.

(10 Punkte)

Aufgabe 3 (Pfadlängen)

Erweitern Sie die Klasse **BinarySearchTree** um folgende private Methoden:

getNumberOfNodes(BinarySearchTreeNode<K> root): Diese Methode berechnet die *Anzahl von inneren Knoten* des binären Suchbaums mit Wurzel **root**.

getInternalPathLength(BinarySearchTreeNode<K> root): Diese Methode soll zu einem gegebenen Binärbaum T mit Wurzel **root**, die *inneren Pfadlänge* $P(T) = \sum_{v \in T}(\text{Tiefe}(v) + 1)$ rekursiv berechnen (siehe auch das aktuelle Präsenzübungsblatt).

getAverageSearchPathLength(BinarySearchTreeNode<K> root): Diese Methode berechnet die *durchschnittliche Suchpfadlänge* $\overline{P(T)} := \frac{1}{|T|} \sum_{v \in T}(\text{Tiefe}(v) + 1)$ des Binärbaums T mit Wurzel **root**, wobei $|T|$ die Anzahl der Knoten in T bezeichnet (siehe auch das aktuelle Präsenzübungsblatt).

getMinSearchPathLength(BinarySearchTreeNode<K> root): Diese Methode berechnet die *minimale Suchpfadlänge* des Binärbaums mit Wurzel **root**.

getMaxSearchPathLength(BinarySearchTreeNode<K> root): Diese Methode berechnet die *maximale Suchpfadlänge* des Binärbaums mit Wurzel **root**.

Schreiben Sie anschließend zu jeder der obigen Methoden eine öffentliche parameterfreie Variante zur jeweiligen Berechnung auf dem gesamten Binärbaum.

(10 Punkte)

Aufgabe 4 (Optimierte Reihenfolge für Einfügeoperationen)

Wie Sie in Aufgabe 1 gesehen haben, hängt die Struktur und somit auch die Laufzeit der Einfüge- und Suchoperationen eines binären Suchbaums von der Reihenfolge der Einfügeoperationen ab.

Erweitern Sie die Klasse **BinarySearchTree** um eine Methode **insertOptimizedKeys(K[] keys)**, welche die im Array **keys** übergebenen Schlüsselwerte in einer optimierten Reihenfolge in den Baum einfügt (d.h. die durchschnittliche Suchpfadlänge soll möglichst klein sein), sowie um eine Methode **insertWorstCaseKeys(K[] keys)**, welche die Schlüsselwerte in schlechtmöglichster Reihenfolge einfügt (d.h. die durchschnittliche Suchpfadlänge soll möglichst groß sein).

Testen Sie anschließend Ihre Implementierung, indem Sie eine Klasse **BinarySearchTreeTest** schreiben, in deren **main**-Methode

1) drei Instanzen **tree1**, . . . ,**tree3** der Klasse **BinarySearchTree<Integer>** erzeugt werden,
2) unter Verwendung von **DataSource4.getRandomData(1000,10000)** (zu finden auf der Vorlesungsseite) ein Array **testData** mit Zufallswerten gefüllt wird
3) und dann die Testdaten mittels der Methoden **insertKeys(..)**, **insertOptimizedKeys(...)** und **insertWorstCaseKeys(..)** in die Bäume **test1**, **test2** bzw. **test3** eingefügt werden.

Geben Sie schließlich zu jedem der drei Bäumen, die

- innere Pfadlänge,
- minimale, mittlere und maximale Suchpfadlänge

aus.

(12 Punkte)

Viel Spaß und viel Erfolg!