Prof. Dr. V. Linnemann
Nils Höller
Universität zu Lübeck
Institut für Informationssysteme

Lübeck, den 15. Mai 2009

Algorithmen und Datenstrukturen

Sommersemester 2009

Sondervorlesung

Aufgabe 1: Implementierung von B-Bäumen

Entwickeln Sie eine Java-Anwendung zur Verwaltung von B-Bäumen beliebiger Ordnung. Auf der Web-Seite der Übung wird Ihnen die Klasse `BBaum` zur Verfügung gestellt, die einen B-Baum teilweise implementiert.

Die Baumknoten werden durch die Klasse `BKnoten` repräsentiert. Das folgende Listing zeigt einen Teil dieser Klasse:

```
import java.util.*;
public class BKnoten {
    private int ordnung;
    private BBaum baum;
    private BKnoten vater;
    private List schluessel;
    private List kinder;
    public BKnoten(BBaum baum, BKnoten vater, int ordnung) {
        this.baum = baum;
        this.vater = vater;
        this.ordnung = ordnung;
        this.schluessel = new ArrayList();
        this.kinder = new java.util.ArrayList();
    }
    // ...
}
```

In einem Knoten werden die Schlüssel in der Liste `schluessel` gespeichert. Die in den Knoten gespeicherten Schlüssel müssen das Interface `Comparable` implementieren. Die Verweise auf Kinderknoten befinden sich in der Liste `kinder`. Ist der Knoten ein Blatt, dann ist die Liste `kinder` leer. Die Variable `baum` verweist auf das Baum-Objekt, zu dem der Knoten gehört. Die Variable enthält die Referenz auf den Vaterknoten bzw. `null` wenn der Knoten die Wurzel des B-Baumes ist.

a) Implementieren Sie die Methode `public Comparable suchen(Comparable key)` in `BKnoten`. Diese Methode soll rekursiv in dem Knoten und in den Nachfahren nach dem gegebenen Schlüssel suchen. Ist der Schlüssel im Baum enthalten, wird das im Baum gespeicherte Objekt zurückgegeben. Andernfalls gibt die Methode `null` zurück.

b) Schreiben Sie eine Hilfsmethode `int einfuegePosition(Comparable key, List ls)` in Klasse `BKnoten`. Diese Methode berechnet die Stelle, in die der Schlüssel `key` in die sortierte Liste `ls` eingefügt werden muss, damit die Ergebnisliste sortiert bleibt. Der Rückgabewert ist also eine Zahl i mit $0 \leq i \leq ls.size()$. Wenn z.B. `ls` die Elemente $20, 30, 40$ enthält, dann ist die Einfügeposition 0 für Schlüssel kleiner als 20; sie ist 1 für Schlüsselwerte zwischen 20 und 30 usw.

c) Wenn ein Schlüssel in einen B-Baum eingefügt werden soll, muss zunächst festgestellt werden, in welchen Teilbaum dieser Schlüssel einzufügen ist. Implementieren Sie zu diesem Zweck eine Methode `BKnoten getTeilbaumFuer(Comparable key)` in `BKnoten`, die für einen Nicht-Blatt-Knoten den Kindknoten zurückgibt, der die Wurzel des Teilbaumes bildet, in den der Schlüssel `key` einzufügen ist.

Aufgabe 2: Implementierung von B-Bäumen

In dieser Aufgabe werden wir die B-Baum-Implementierung erweitern.

a) Zur Behandlung der Überläufe beim Einfügen wollen wir den einfachen Einfügealgorithmus verwenden: wenn ein Knoten zu viele Schlüssel enthält, wird er in der Mitte aufgespalten. Der Schlüssel in der Mitte wird ”nach oben´´ (zum Vaterknoten) verschoben. Die Schlüssel und ggf. die Kinderknoten links und rechts von diesem Schlüssel bilden jeweils einen neuen Knoten. Der ursprüngliche Knoten wird durch diese beiden neuen Knoten ersetzt.

   Implementieren Sie eine Methode `void split()` in `BKnoten`, die diesen Algorithmus implementiert. Achten Sie auf den Sonderfall, dass der Knoten mit Überlauf die Wurzel des Baumes ist.

b) Implementieren Sie die Methode `public boolean einfuegen(Comparable key)` in `BKnoten`. Diese Methode soll den gegebenen Schlüssel in die richtige

Stelle im Knoten oder in einem der Nachfahrknoten speichern. Ist der Schlüssel schon im Baum enthalten, gibt die Methode `false` zurück, andernfalls `true`. Verwenden Sie zur Behandlung der Überläufe die Methode `split()` vom letzten Aufgabenteil. Nutzen Sie auch die im letzten Aufgabenblatt realisierten Methoden, um den Knoten zu bestimmen, in den der neue Schlüssel einzufügen ist.