Prof. Dr. Stefan Brass
Institut für Informatik
MLU Halle-Wittenberg

04. Dezember 2015

Datenbanken II B: DBMS-Implementierung
— Hausaufgabe 7A —

In dieser Aufgabe soll das Implementierungs-Projekt fortgesetzt werden.

a) Legen Sie eine Datei `btype.h` mit einem Aufzählungstyp `btype_t` für Blocktypen an. Man braucht mindestens die Werte

   - `BTYPE_EMPTY` für nicht belegte Blöcke,
   - `BTYPE_FILE_HEADER` für den ersten Block der Datei mit Basisdaten der Datenbank (z.B. Freispeicher-Verwaltung) und
   - `BTYPE_REL_DATA` für Blöcke mit Relationen-Daten.

   Wenn man niemals geschriebene Blöcke erkennen will, könnte es nützlich sein, den ersten Wert (0) z.B. als `BTYPE_INVALID` zu definieren. Falls man das Programm später erweitern will, würde es noch weitere Arten von Blöcken geben, z.B. Blöcke für B-Baum Indexe, und Blöcke für Relationen mit variabler Zeilenlänge. Das werden wir aber in den Hausaufgaben zu dieser Vorlesung nicht mehr schaffen.

b) Definieren Sie nun eine Klasse `block_c` für Datenbank-Blöcke (in der Datei `block.h`). Diese Klasse wird später Unterklassen für die verschiedenen Arten von Blöcken haben. In `block.h` definieren Sie bitte auch eine Konstante `BLOCK_SIZE` für die Blockgröße als `8192` (8 KByte). In meinem früheren Programm sind Objekte der Klasse `block_c` genau `BLOCK_SIZE` Bytes groß (sie enthalten insbesondere ein Array von Zeichen für die eigentlichen Daten, mit einer `union` überlagert mit `short` und `int`-Werten). Die Unterklassen fügen dann neue Methoden hinzu, aber keine neuen Attribute. Das bedeutet natürlich, dass man die Datenstrukturen in den Blöcken auf relativ tiefer Implementierungsebene aufbauen muss (es werden Konstanten und Makros für die Berechnung von Offsets im Daten-Array definiert).

   Man könnte natürlich auch anders vorgehen, und einen Block zunächst wesentlich kleiner als 8 KByte sein lassen. Die Lese- und Schreib-Routinen werden aber immer 8 KByte zwischen Hauptspeicher und Platte bewegen. Eventuell braucht man ein Container-Objekt der vorgegeben Größe, in das man die tatsächlich benötigten Daten als kleineres Objekt hineinlegt (mit einem Typ-Cast oder einer Union).

   - Die Klasse `block_c` muss mindestens eine Methode `type()` haben, die den Typ des jeweiligen Blocks liefert (also einen Wert vom Typ `btype_t`).
   - Der Konstruktor soll einen leeren Block erzeugen, vom Typ `BTYPE_EMPTY` und mit definiertem Inhalt (alles 0).

- Es ist zu empfehlen, dass man ausserdem die Korrektheit des Blocks mit einer Methode `check_err()` prüfen kann (das Ergebnis könnte eine Fehlermeldung als String sein, oder ein Null-Zeiger im positiven Fall). Man könnte z.B. in jeden Block am Anfang und eventuell auch am Ende ein bestimmtes Bitmuster schreiben ("Magic Number"), an dem erkannt wird, dass es ein Datenblock von diesem Programm ist.
- Es könnte später nützlich sein, wenn man auch die Nummer des Blockes in der Datei im Block-Objekt speichert. Dafür würde ich die Methode `block_no()` zum Abfragen der Blocknummer und `set_block_no()` zum Setzen der Blocknummer vorgesehen. Die Blocknummer ist ein 32-Bit Wert (Datenbanken können groß sein). Man könnte sie `unsigned` machen, muss dann aber später öfters mit Typ-Inkompatibilitäten kämpfen. Ich habe in einem früheren Programm einen Typ `bno_t` (in `bno.h`) für Blocknummern definiert, aber möglicherweise ist das übertrieben.
- Ein üblicher Trick, um nur partiell geschriebene Blöcke zu erkennen, ist es, ein Bitmuster am Anfang und am Ende eines Blockes abzulegen, das bei jedem Schreibvorgang invertiert wird. Die Bitmuster müssen dann immer gleich sein. Wenn Sie das machen wollen, können Sie eine Methode `prepare_write()` vorsehen, die diese Bitmuster invertiert, und ein Mal vor jedem Schreibvorgang aufgerufen wird.

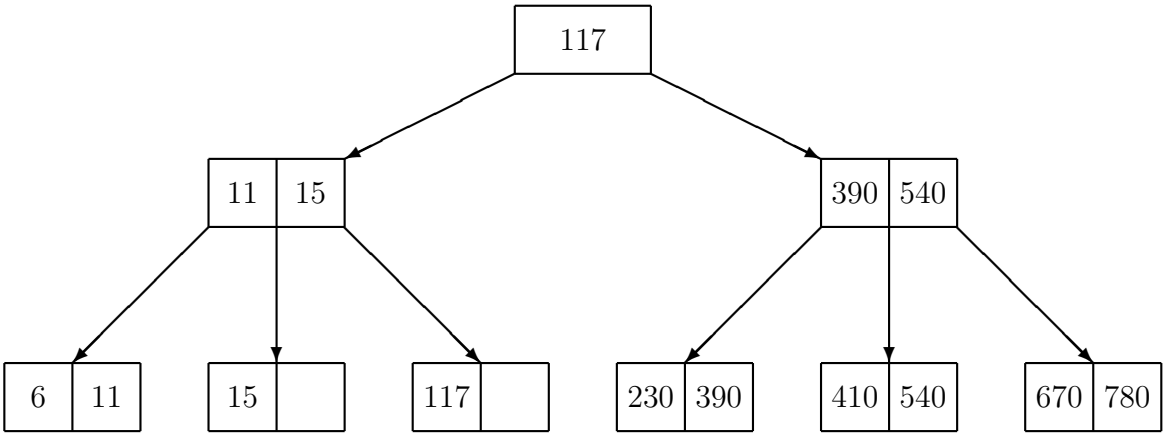
c) Die Objekte Ihres Puffer-Managers müssen dann auf einen solchen Block verweisen, oder einen Block als Attribut enthalten. Passen Sie also Ihre Klasse `buf_c` entsprechend an.

Auf dem nächsten Übungsblatt werden die Blöcke dann wirklich von einer Datei gelesen. Ziel ist es, wenigstens eine einfache Relation mit n `int`-Spalten zu implementieren (also mit Zeilen fester Größe). Möglicherweise werden wir aber auch noch eine einfache Indexstruktur implementieren.

— Hausaufgabe 7B —

Hinweis: Beachten Sie bitte, dass die Studienleistung ganz ohne Programmierung nicht zu erreichen ist.

Gegeben sei folgender B^+-Baum der Höhe 3 und Ordnung 1 (minimal 1 Eintrag pro Knoten, maximal 2):



Geben Sie den resultierenden B^+-Baum jeweils nach den folgenden 4 Operationen an. Wenden Sie die Operationen immer auf den oben angegebenen B^+-Baum an, nicht auf das Ergebnis des vorhergehenden Schrittes.

a) Einfügen von 13.

b) Einfügen von 9.

c) Einfügen von 290.

d) Löschen von 15.