Introduction to Computer Science II
Grundlagen der Informatik II

June 11, 2014

Lab 4 - B-Trees

Deadline: June 22, 2014 @ 23.59

Submission System:
https://gdi2.cdc.informatik.tu-darmstadt.de/cgi-bin/index.cgi

Version 1.0

1 Introduction

For this lab, we consider the same library system using RFID tags as in lab 1 (QuickSort). This time, the values read by the readers placed in the library are not stored in a central file but in a B-Tree structure to simplify the management of the data. The B-Tree structure implements basic operations such as insert, find and delete of entries. The format of the entries inserted in the B-Tree is the same as defined in lab 1:

```
Book_serial_number;ReaderID;STATUS
```

The `Book_serial_number` and the `ReaderID` together form the *key* of the entries in the B-Tree structure (the key is the string, which results from the concatenation of these 2 strings). The `STATUS` represents the *data* stored in each entry.

2 Task

Your task is to develop a Java class (`B_Tree`) which implements a B-Tree structure as introduced in the lecture to store the information read by the RFID readers in the library. A B-Tree is a height-balanced tree, i.e., each path from the root to a leaf has the same length. Each node in a B-Tree with minimum degree t (except for the root) has at least t and at most $2t$ children and thus at least $t-1$ and at most $2t-1$ keys. The root of the B-Tree is either a leaf or has at least 2 children. In the B-Tree the keys and data are stored in both internal nodes and in the leaves. The keys stored in the nodes of the B-Tree are unique. The minimum degree of the B-Tree cannot be changed after its initialization, i.e., t cannot be changed once the B-Tree is initialized.

3 The Code

We provide the following Java classes and test files within the zip archive **Lab4.zip** which contains the Java project with the building blocks. The classes are split in two packages – the 'frame' and the 'lab' package. You should implement your solution starting with the skeleton code provided in the zip archive.

3.1 The 'frame' package

The submission system will use its own copy of these classes when testing your source code. Changes made by you within this package will not be considered by the submission system.

3.1.1 `AllTests.java`

This is the JUnit test case class. The test cases defined by this class are used to test the correctness of your solution.

3.1.2 `TestNode.java`

This class is used to represent nodes during testing.

3.1.3 `DotFileConstants.java`

Contains building blocks of the dot language.

3.1.4 `Entry.java`

The entries of the B-Tree are stored in objects of type `Entry`. This class is given and implements the following interfaces. Modification to the class `Entry` are not allowed.

- `EntryInterface`

  This interface defines the following methods:

  - `void setKey(String newKey)`: This method sets the value of the key of the entry to the given value *newKey*
  - `void setData(String newData)`: This method sets the value of the data of the entry to the given value *newData*
  - `String getKey()`: This method returns the value of the key of the entry
  - `String getData()`: This method returns the value of the data of the entry
  - `String toString()`: This method returns the string representation of the entry. The string representation has the following form

    `Book_serial_number;ReaderID;STATUS (e.g., Z8IG4;LDXS;OK)`

- `Comparable<Entry>`

  This interface is needed to be able to compare two objects of type `Entry` directly (see JAVA documentation for details).

3.1.5 `EntryInterface.java`

Defines the interface 'EntryInterface'.

3.2 The 'lab' package

This package contains the files you are allowed to modify. You are free to add additional classes (in new files within the lab packages, no subpackages), as well as to add any additional methods or variables to the provided classes, as long as you **do not change the signature (*name, parameter type and number*) of any given method**, as they will be used by the JUnit tests. Any source code changes that you make outside the 'lab' package will be ignored when you upload your source code to the submission system.

3.2.1 B_Tree.java

For the `B_Tree` class the following methods are to be implemented:

- `public B_Tree(int t)`

  The constructor of the class. It takes the integer t as input, which represents the minimum degree of the the B-Tree structure. t cannot be changed once a `B_Tree` object is created.

- `public int constructB_TreeFromFile (String filename)`

  This method takes as input the name of a file containing a sequence of entries that should be inserted to the B-Tree in the order they appear in the file. You can assume that the file is located in the same directory as the executable program. The input file is similar to the input file for lab 1. The return value is the number of entries *successfully* inserted into the B-Tree.

  Note: The entries of the B-Tree are stored in objects of type `Entry` (cf. Section 3.1.4).

  A sample input file is shown below (see lab 1 for more details).

  ```
  Z8IG4;LDXS;OK
  OX6F9;ERSY;OK
  YSI7Q;ERSY;OK
  6C8IV;ERSY;Error
  YSI7Q;4O09;OK
  EMBXP;GQ9Y;OK
  5MXGT;7L8Q;Error
  FOC9U;7L8Q;OK
  XOH3X;GQ9Y;Error
  XOH3X;ERSY;Error
  XDYF6;P80S;OK
  GFN81;7L8Q;Error
  FOC9U;7L8Q;OK
  WN178;GQ9Y;OK
  ```

  To insert these entries in the B-Tree structure, you should use the following method:

- `public boolean insert(Entry insertEntry)`

  This method inserts the entry *insertEntry* in the right place into the B-Tree. Note that you have to deal with overflows in this method, e.g., if you want to insert an entry into a leaf which already contains $2t - 1$ entries. This method returns *true* if the insertion of the entry *insertEntry* is successful and $false$ if the *key* of this entry already exists in the B-Tree.

- `public Entry delete(String deleteKey)`

  This method deletes the entry from the B-Tree structure, having *deleteKey* as key. In this method you have to distinguish between two cases:

  1. The entry, having *deleteKey* as key, is located in a leaf.
  2. The entry, having *deleteKey* as key, is located in an internal node.

  This method returns the entry, having *deleteKey* as key if the deletion is successful and *null* if the key *deleteKey* is not found in any entry of the B-Tree.

  Please note that the B-Tree is not always unique after deletion of one element. Therefore you are requested to follow these instructions while implementing the *delete* method:

  1. You should always try to do rotation before merging. If rotation is not possible, merging is done.
  2. Try always first to rotate with the left neighbor.
  3. Try always first to merge with the left neighbor.
  4. If you want to delete an entry from an internal node, replace this entry with the one having the next bigger key than the deleted one.

- `public Entry find(String searchKey)`

  This method searches in the B-Tree for the entry with key *searchKey*. It returns the entry, having *searchKey* as key if such an entry is found, *null* otherwise.

- `public ArrayList<String> getB_Tree()`

  This method returns a `ArrayList<String>` containing the output B-Tree. The output should be directly interpretable dot code as described in section 4. Each item in the `ArrayList` corresponds to one line of the output tree. So for the B-Tree in Figure 4 the output ArrayList would be of length 12 as numbered in Figure 3. The nodes of the output tree should only contain the keys of the entries and not the data.

- `public int getB_TreeHeight()`

  This method returns the height of the B-Tree. If the B-Tree is empty or only contains the root node this method should return 0.

- `public ArrayList<Entry> getInorderTraversal()`

  This method performs an in-order traversal of the B-Tree and adds each entry to a `ArrayList<Entry>`. Thus, the returned ArrayList contains the entries of the B-Tree in ascending order.

- `public int getB_TreeSize()`

  This method returns the number of entries in the B-Tree (not the number of nodes). For example, for the B-Tree in figure 4, this method should return 13.

3.2.2 B_TreeNode.java

This class can be used to implement nodes of the B-Tree that store the entries. You are given the flexibility to either utilize this class or not.

3.3 Test Files

We provide one input file for testing as follows:

- TestFile.txt

You are encouraged to test your solution using additional input files as well. Note that you need to write your own JUnit test cases in order to run with your customized input graphs. Think about the assumptions made on the input. To make sure that your solution works, do test it with all the test cases provided. Apart from the given input files, the submission system will test your solution with several additional input files, to confirm the correctness of your program.

4 Output format: dot

As output format for this lab we will use a subset of the `dot` language introduced in Lab 2 (Navigation). This subset is described below. We provide you in this section with an example to understand the part of the dot language you need for this lab. Therefore we use in this example for simplicity integers as keys and not strings.

4.1 dot language

The representation of the B-Tree node in Figure 2 in dot language is provided in Figure 1. The boxes labeled with "*" in the node representation are the pointers to the nodes of the next level in the B-Tree structure. $f0$, $f1$, $f2$, ... are the names of the boxes in the node representation.

```
Digraph{
node[shape=record];
node1[label="<f0>*|<f1>42|<f2>*|<f3>45|<f4>*|<f5>77|<f6>*"];
}
```

Figure 1: The code for Figure 2

4.2 Example

The graph in Figure 4 represents an example of a B-Tree with minimum degree 3, where the keys of the entries are integers. The B-Tree results from inserting the following integers in the given order: 11, 18, 43, 47, 42, 31, 55, 62, 71, 77, 83,

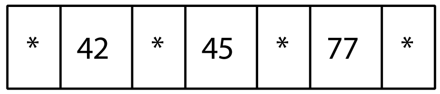

Figure 2: An example of a B-Tree node

91, 99. Figure 3 represents the corresponding code in dot language. All output trees should follow the format in Figure 3, i.e., leaves look like internal nodes, but have no pointers set to other nodes. The ordering of the entries in the dot representation is irrelevant except for the first two and the last one. The format of the nodes of the output B-Tree should only contain the keys and no data. The root node of a B-Tree in this lab should be explicitly named *root* as in figure 3. To draw a pointer from a node to another one in the next level, you need to specify the name of the box containing the pointer like in figure 3 (e.g., `root:f0->node2;`).

```
Digraph{
node[shape=record];
root[label="<f0>*|<f1>42|<f2>*|<f3>55|<f4>*|<f5>77|<f6>*"];
node2[label="<f0>*|<f1>11|<f2>*|<f3>18|<f4>*|<f5>31|<f6>*"];
node3[label="<f0>*|<f1>43|<f2>*|<f3>47|<f4>*"];
node4[label="<f0>*|<f1>62|<f2>*|<f3>71|<f4>*"];
node5[label="<f0>*|<f1>83|<f2>*|<f3>91|<f4>*|<f5>99|<f6>*"];
root:f0->node2;
root:f2->node3;
root:f4->node4;
root:f6->node5;
}
```

Figure 3: Example code for a B-Tree in dot language

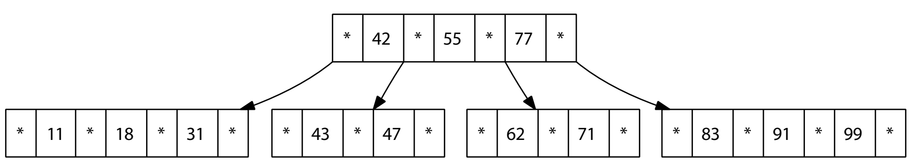

Figure 4: A simple B-Tree with $t = 3$ described by the code in Figure 3