Assignment 3

COS 212

Department of Computer Science
Deadline: 29 May 2020 at 23:00

Objectives:

- Implement a B^{+}-Tree.
- Use the B^{+}-Tree in a simplistic database table implementation.

General instructions:

- This assignment should be completed individually, **no group effort** is allowed.
- Be ready to upload your assignment well before the deadline, as no extension will be granted.
- You may not import any of Java's built-in data structures. Doing so will result in a mark of zero. If you require additional data structures, you will have to implement them yourself.
- If your code does not compile you will be awarded a mark of zero. Only the output of your program will be considered for marks, but your code may be inspected for the presence or absence of certain prescribed features.
- **All submissions will be checked for plagiarism.**
- Read the entire assignment before you start coding.
- You will be afforded three upload opportunities.

Plagiarism:

The Department of Computer Science considers plagiarism as a serious offence. Disciplinary action will be taken against students who commit plagiarism. Plagiarism includes copying someone else's work without consent, copying a friend's work (even with consent) and copying material (such as text or program code) from the Internet. Copying will not be tolerated in this course. For a formal definition of plagiarism, the student is referred to `http://www.library.up.ac.za/plagiarism/index.htm` (from the main page of the University of Pretoria site, follow the *Library* quick link, and then choose the *Plagiarism* option under the *Services* menu). If you have any form of question regarding this, please ask one of the lecturers, to avoid any misunderstanding. Also note that the OOP principle of code re-use does not mean that you should copy and adapt code to suit your solution.

After completing this assignment:

Upon successful completion of this assignment you will have implemented a B$^+$-Tree. You will also have implemented your own simplistic database table that manipulates its data using your B$^+$-Tree based database indexes.

Task 1

Consider the STUDENT database table in *Table 1.* The first column is a unique internal identifier given by the database to each record in a table. This column is hidden from all general database activity. All other columns in this table represent user data that is visible. This particular table contains student information consisting of student number, name and surname. The student number has been designated a primary key index since it is unique at a university. To speed up searching for students by surname, another index has been created for the surname column.

	PrimKey		**SecKey**
RowID	**StudentID**	**Name**	**Surname**
1	16230943	Lerato	Molefe
2	17248830	Isabel	Muller
3	16094340	John	Botha
4	17012340	Michael	Evans

Table 1: A STUDENT database table

Database indexes are generally implemented using B$^+$-Trees. B$^+$-Trees represent an efficient way to store indexes in memory and provide a fast and constant way to search for records stored on disk. The two indexes of *Table 1* can be stored in two B$^+$-Trees. The first tree would use the student numbers as keys, while the second one would use the surnames. In both trees, the row id would be the value that is stored in the leaf nodes.

Your task is to implement a B$^+$-Tree that can be used to store both the indexes that are defined in the STUDENT table in *Table 1.* Download the archive `assignment3CodeTask1.zip`. This provides a functional B$^+$-Tree class and partially implemented B$^+$-Tree node subclasses to use. You will have to complete the `BPTreeNode` class by implementing the insert, search, delete and values methods. The classes `BPTreeInnerNode` and `BPTreeLeafNode` inherit from the `BPTreeNode` parent class and will contain type specific functionality.

You should use your own helper functions to assist in implementing the insert, search, delete and values methods as per the specification documented in the code. However, you may not modify any of the given members, methods or method signatures.

You are also provided with the file `Main.java`, which will test some code functionality. It also provides an example of how the indexes of *Table 1* could be defined and used. You are encouraged to write your own test code in this file. Test your code thoroughly using this file before submitting it for marking.

Search

Searching can be performed either sequentially via the linked leaf nodes or by tree traversal. The value associated with the given key should be returned. If the given key is not found, `null` should be returned.

Deletion

Deletion will always take place at the leaf node level. A possible corresponding separator key in the parent internal node should not be removed during deletion. Only a future node merge can lead to the separator key removal.

Underflow

If a node underflows, first check if the left sibling has enough keys to share. If it doesn't, check if the right sibling can share keys. If neither of the siblings have enough keys to share, then the underflowing node merges with its sibling (left sibling if possible, otherwise right sibling).

Merging

Merging of nodes will require the update of the separator keys in the parent internal node. Firstly, old separator keys will need to be removed. Secondly, a new separator key may need to be added for the newly merged node.

Example

You can use the B$^+$-Tree visualiser (`https://www.cs.usfca.edu/~galles/visualization/BPlusTree.html`) to get a visual demonstration as to how the insertion and deletion strategies should work. Unfortunately, the visualiser does not leave the separator key in an internal node once the key has been deleted from the leaf node. You should not follow this strategy in your implementation, but stick to the strategy used in the textbook.

Task 2

Now consider a simplistic implementation of a database table that makes use of the B$^+$-Tree implementation created in task 1. A `Table` class stores its row data in an array of records. A `Record` class is used for that purpose and it stores an array of values and returns them as a string. Another array in the `Table` class stores the table columns that are used in the records. The `Table` class also stores an array of indexes that apply to specific record columns. An `Index` class is used for that purpose which contains the B$^+$-Tree and some other information.

Your task is to implement a database table that can be used to store both the data and indexes that are defined in the STUDENT table in *Table 1*. This database table should also support the manipulation of its data using its available indexes with the standard SQL methods select, insert and delete.

Download the archive `assignment3CodeTask2.zip`. This provides a partially implemented Table class and functional helper classes to use. You will have to complete the `Table` class by implementing the SQL insert, search and delete methods, as well as the index create and print methods as per the specification documented in the code.

To standardise output for marking purposes, an `Error` class has also been provided that contains standard messages that should be used as documented in the code. To compile the provided code, you need to add your completed files from task 1.

You should use your own helper functions to assist in implementing the insert, search and delete SQL query methods as well as the index create and print methods. However, you may not modify any of the given members, methods or method signatures.

You are also provided with the file `Main.java`, which will test some code functionality. It provides an example of how the data and indexes from *Table 1* could be defined and used. You are encouraged to write your own test code in this file. Test your code thoroughly using this file before submitting it for marking.

Duplicates

Since the B$^+$-Tree search method only returns a single value, returning or removing multiple rows using the index is not possible without modification. Therefore, the select and delete methods only need to return or delete the first matching record found.

Matching

The select and delete methods only need to make provision for exact matches of the where clause. Special syntax that allows partial matches normally found in SQL does not need to be supported.

Sizing

The `Table` class constructor makes provision for an initial number of records and indexes. You may assume testing will stay within these limits and your insert() and createIndex() methods do not need to support increasing the initial array size when full. The B$^+$-Trees used to store the index data should be of order 4.

Submission instructions

You need to create your own makefile and submit it along with your Java code. Your code should be compiled with the following command:

```
javac *.java
```

Your makefile should also include a `run` rule which will execute your code if typed on the command line as in `make run`.

For your submission, you need to place all your source files including your makefile in a zip or tar/gzip archive (you need to compress your tar archive) named uXXXXXXXX.zip or uXXXXXXXX.tar.gz where XXXXXXXX is your student/staff number. There should be no folders/subfolders in your archive.

Submit your code for marking under the appropriate links (*Assignment 3: Task X*) before the deadline. Separate upload links are provided for the two tasks. Each link grants 3 uploads. Since every upload is a marking opportunity, do not use it to test your code.