# Assignment 3

# COS 212

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA
Denkleiers • Leading Minds • Dikgopolo tša Dihlalefi

Department of Computer Science
Deadline: 4 June 2021 at 23:00

## Objectives:

- Implement a $B^+$-Tree with key sharing and duplicate handling.
- 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.

## Part 1 [140]

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 |
| 5 | 17325368 | Daniel | 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 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 *RowID* 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*. Your B$^+$-Tree should work with generic types for both its keys and its values (although the values will always be `Integer` *RowIDs* for this assignment). You have been provided a functional B$^+$-Tree class and partially implemented B$^+$-Tree node subclasses to use in the *Part 1* folder alongside this specification. 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 have also been given a `ValueNode` class which is described in the *Duplicate Handling* section below. Please observe the given code specification for further details.

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.

### Duplicate Handling

If column does not have unique values and it is used to index a B$^+$-Tree, the B$^+$-Tree would typically contain duplicate keys. Duplicate keys decrease the efficiency of the B$^+$-Tree search operation. In

order to cater for this, if multiple values are inserted for a particular key, the values are prepended to the beginning of a linked list stack in the leaf node corresponding to the particular key. Therefore, each **leaf node** stores an array of keys as well as a corresponding array of type `ValueNode`. Each `ValueNode` in the array is a head node of a linked list which contains the value(s) for each particular key. In this way, the tree can contain multiple different values corresponding to a particular key. Furthermore, a linked list can easily expand when more duplicates are added. Figure 1 visually demonstrates how duplicate keys are handled in the $B^+$-Tree.

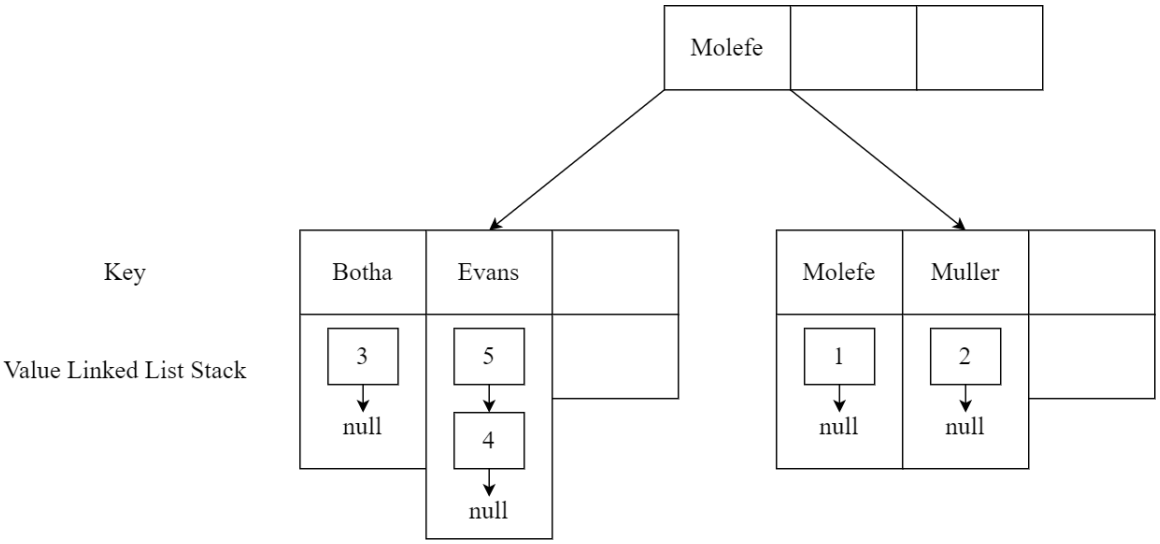


Figure 1: $B^+$-Tree of order 4 where values are inserted in the same order as given in Table 1. The surname is used as a key and the values are the *RowIDs* which are stored in linked lists in the leaf nodes. Notice how "Evans" is a duplicate key for two different values. Also notice that the value 5 is inserted after value 4 but it is the head of the list because new `ValueNodes` are prepended for duplicate keys.

## Search

Searching can be performed either sequentially via the linked leaf nodes or by tree traversal. The `ValueNode` associated with the given key should be returned (which is the head of the linked list corresponding to the given key). 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. When a key is removed from a leaf, **all** of the corresponding value(s) relating to that key are also removed (i.e. the head of the list for that key is set to `null` or overwritten depending on the shifting of keys).

## 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). Remember that siblings share the same parent node.

### 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.

### Key Sharing

In this assignment, keys are only shared when a node underflows (i.e. after a deletion operation). Keys are not shared when a node overflows after an insertion operation. If a node overflows, then it must be split. Please observe the extensive example given in the **Main.java**.

### 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 described in this specification.

### Submission Part 1

You need to submit your source files on the Fitch Fork website (https://ff.cs.up.ac.za/). All methods need to be implemented (or at least stubbed) before submission. Place your **BPTree.java**, **BPTreeInnerNode.java**, **BPTreeLeafNode.java**, **BPTreeNode.java**, **ValueNode.java** file in a zip archive named uXXXXXXXX.zip where XXXXXXXX is your student number. There is no need to include any other files in your submission. You have 3 submissions and your best mark will be your final mark. Do not use Fitch Fork to test your code because you have limited marking opportunities. Upload your archive to the *Assignment 3 Part 1* slot on the Fitch Fork website. Submit your work before the deadline. **No late submissions will be accepted!**

## Part 2 [100]

Now consider a simplistic implementation of a database table that makes use of the B$^+$-Tree implementation created in Part 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 column 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 for a given table. The provided test code creates a music collection table. This database table should also support the manipulation of its data using its available indexes with the standard SQL methods select, insert and delete.

Please see the `Part 2` folder given with this specification. 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**. The code has comments which precisely outline what is required.

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. Use a `System.out.println()` to

print the error messages when required. **To compile the provided code, you need to add your completed files from Part 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 can be used on a music collection. You are encouraged to write your own test code in this file. Test your code thoroughly using this file before submitting it for marking.

## 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 the order provided to the constructor of the `Table` class.

## Assumptions

- Records will only store strings, integers or a combination of the two.
- The number of columns in a record will not exceed 100.

## Submission Part 2

You need to submit your source files on the Fitch Fork website (https://ff.cs.up.ac.za/). All methods need to be implemented (or at least stubbed) before submission. Place your **BPTree.java**, **BPTreeInnerNode.java**, **BPTreeLeafNode.java**, **BPTreeNode.java**, **ValueNode.java**, **Error.java**, **Index.java**, **Record.java**, **Table.java** file in a zip archive named uXXXXXXXX.zip where XXXXXXXX is your student number. There is no need to include any other files in your submission. You have 3 submissions and your best mark will be your final mark. Do not use Fitch Fork to test your code because you have limited marking opportunities. Upload your archive to the *Assignment 3 Part 2* slot on the Fitch Fork website. Submit your work before the deadline. **No late submissions will be accepted!**