# Introduction to Database (Spring 2024)

# Homework #4 (80 Pts, Due Date: May 27)

**Student ID** ____________________

**Name** ____________________

**NOTE**:

✓ Compress ‘main.c’, ‘B+TREE.c’, ‘B+TREE.h,’ and ‘your report’ (this current document file in PDF format) and submit with the filename ‘HW4_STUDENT ID.zip’. The directory structure should look like this:

```
HW4_2023123456.zip
 ├ main.c
 ├ B+TREE.c
 ├ B+TREE.h
 └ HW4_2023123456.pdf (This document in PDF format)
```

✓ You can add/modify existing functions but do not use additional libraries.

✓ You may also refer to the B-tree implementation in (BTREE/BTREE.c) when implementing B+-tree.

✓ The use of AI chatbots (e.g. ChatGPT) or plagiarism will be considered cheating.

**(1)** [**60 pts**] Implement **insertion** and **deletion** operations of B+-tree. You also show the results together for given element sequences. **(Insertion: 20 pts, Deletion: 20 pts)**

**Definition of B+-tree**

1. Every node has at most m children (m: Max. Degree of B+-tree).
2. Every node that contains data is a leaf node.
3. Every node (except root) has at least $\lceil (m+1)/2 \rceil$ children.
4. A non-leaf node with k children contains k−1 keys.
5. All leaves appear on the same level

To implement the B+ tree, please refer to the following sites.

- https://en.wikipedia.org/wiki/B%2B_tree
- https://www.cs.usfca.edu/~galles/visualization/BPlusTree.html
- https://iq.opengenus.org/b-tree-search-insert-delete-operations/

**(a)** Fill in the blank B+-tree template (B+TREE/B+TREE.c). Implement three functions: _insert, _splitChild, and _remove.

See the README.md file for instructions on how to test your B+TREE program.

Here are some tips for each function:

**_insert:**

- _insert function inserts a given key into the appropriate position in the B+ tree. It also triggers the _balancing function if needed to ensure the tree remains balanced.
- If the node is a leaf node, connect the new leaf node with the right node.
- When the node is an index node, the key is passed down to the appropriate child node.

**_splitChild:**

- _splitChild function handles the splitting of a node into two when it becomes full in a B+ tree. If necessary, it creates a new parent node or updates the existing one.
- If a leaf node is full, split into two noes and create a parent node. Note that the middle key goes to the right child. (e.g., [50, 55, 60, 65, 70] → [50, 55], [60, 65, 70])
- When splitting the index node, make sure to move the child nodes to appropriate index nodes.
- The two split nodes should have the same parent node.

**_remove:**

- The _remove function removes the given key from the B+ tree while maintaining the tree's balanced state if necessary.
- It operates recursively, searching the tree until it finds the node with the key to be removed. After removing the key, it calls the _balancingAfterDel function to ensure the tree's balance is preserved.

Answer: Submit your code to i-campus. Don't write your code here.

**(b)** Show the B+-tree for each case.

Answer: Show your results including the snapshot screen.

**Insertion:**

1) Max degree = 3

Insert(1, 3, 7, 10, 11, 13, 14, 15, 18, 16, 19, 24, 25, 26)

2) Max degree = 4

Insert(1, 3, 7, 10, 11, 13, 14, 15, 18, 16, 19, 24, 25, 26)

**Deletion:**

1) Max degree = 3

Insert(1, 3, 7, 10, 11, 13, 14, 15, 18, 16, 19, 24, 25, 26) Remove (13)

2) Max degree = 4

Insert(1, 3, 7, 10, 11, 13, 14, 15, 18, 16, 19, 24, 25, 26) Remove (13)

**(2) [20 pts]** Compare the index scan and full table scan using SQL queries on MySQL. The selectivity of a predicate indicates how many rows from a row set will satisfy the predicate.

$$\text{selectivity} = \frac{Numbers\ of\ rows\ satisfying\ a\ predicate}{Total\ number\ of\ rows} \times 100\%$$

Compare the running time between index scan and full table scan according to different data selectivity and draw the graph to compare two scan methods depending on the selectivity. (Fix the total number of rows as 20,000,000). You also should explain the experimental results.

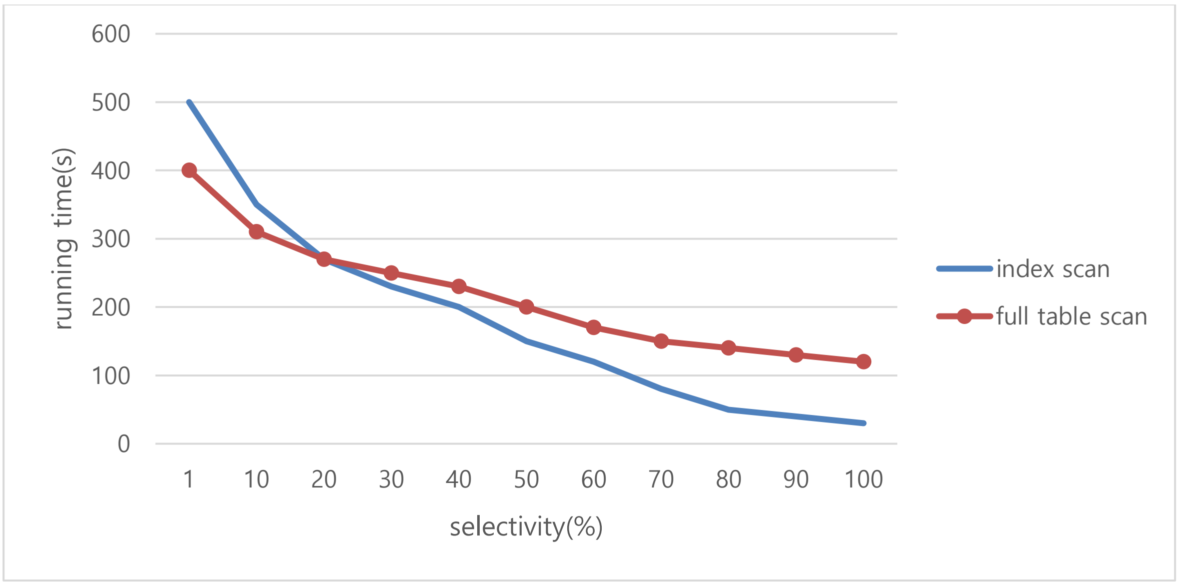
**Example code for generating synthetic table.**

```
/* Make a table */
DROP TABLE TEST;
CREATE TABLE TEST (a INT, b INT);

DELIMITER $$

DROP PROCEDURE IF EXISTS loopInsert $$

CREATE PROCEDURE loopInsert()
BEGIN
	DECLARE i INT DEFAULT 1;
	WHILE i <= 20000000 DO
		INSERT INTO TEST (a, b) VALUES (i, i);
		SET i = i + 1;
	END WHILE;
	COMMIT;
END$$

DELIMITER ;

SET autocommit=0;
CALL loopInsert;
COMMIT;
SET autocommit=1;

/* Make a index */
ALTER TABLE TEST ADD INDEX(a);

/* Compare the running time between index scan and full table scan at selectivity 50% */
SELECT SUM(a)
FROM TEST
WHERE a > 10000000;

SELECT SUM(b)
FROM TEST
WHERE b > 10000000;
```

**(a)** Comparison graph for running time over selectivity.

Please note that this figure is only for example. It is incorrect.

Answer: Show the comparison graph.



**(b)** Explain why index scan is faster or slower than full table scan depending on the selectivity in your comparison results.

Answer: Explain your comparison graph (= answer of (2)-(a)).