

**NANYANG  
TECHNOLOGICAL  
UNIVERSITY**  

---

**SINGAPORE**

**CZ4031**  
**Database System Principles**

**Project 1 Report**  
**Group 4**

**Members:**

Rosario Gelli Ann Labayen (U1822818E)

Tang Heem Yew (U1820996J)

Ngiam Jing Xiang (U1820038H)

Lexx Audrey Pecson Manalansan (U1822109H)

# Introduction

This report aims to design and implement a movie database management system, storage and indexing. The following specifications are set

- Programming language : Java
- Indexing component: B+ tree

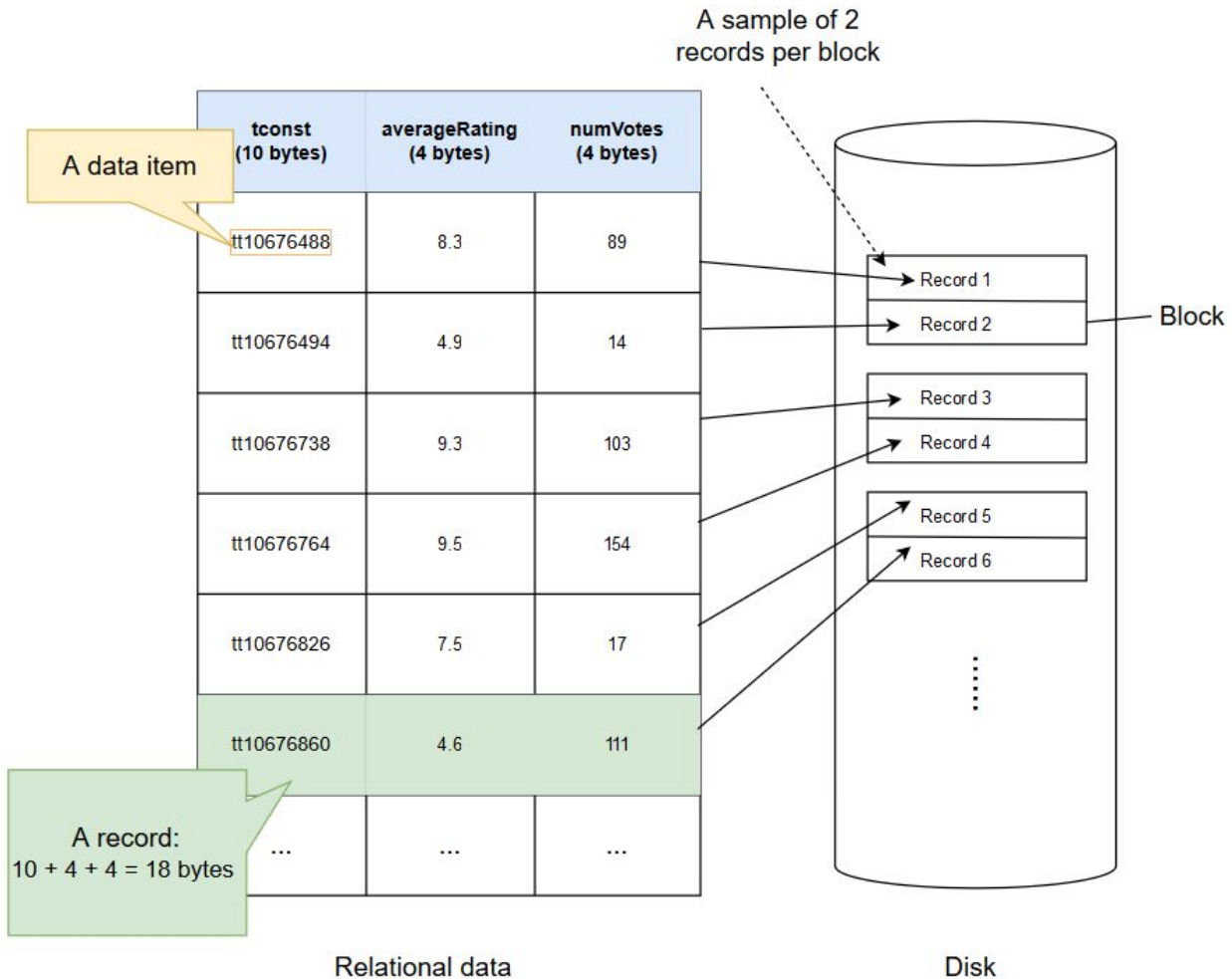
The data contains a movie information list that each consist of tconst (unique movie id), averageRating, and numVotes. The data contains over 1 million records. Table below shows a sample of the data.

tconst	averageRating	numVotes
tt10676488	8.3	89
tt10676494	4.9	14
tt10676738	9.3	103

## Storage Design Consideration

Based on the data list provided, the following design considerations are made for the storage of the components, a class named *Record* will store the following :

Data Column	Data Type	Consideration
<i>tconst</i>	String	String is a suitable data type as it will only take up 10 bytes.
<i>averageRating</i>	float	<p>As decimal is part of the required input for the calculation of the average rating of each movie, and with 10.0 as the possible maximum average rating for a record, float and double can be used.</p> <p>Float can store about 7 decimal digits. <b>(Size : 4 bytes)</b> Double can store about 14 decimal digits. <b>(Size : 8 bytes)</b></p> <p>Given our data that only needs 1 decimal digit, considering the size and the number of decimal digits, float is sufficient for this data.</p>
<i>numVotes</i>	int	<p><i>numVotes</i> refers to the number of votes the title has received. Inputs required for this data are whole numbers.</p> <p>Hence, int with the size of 4 bytes will be the ideal data type for this data.</p>



The format we have decided to use is fixed-length fields. The rationale we choose fixed-length fields is because it is easier to interpret, however some space is wasted in the memory as compared to variable-length fields.

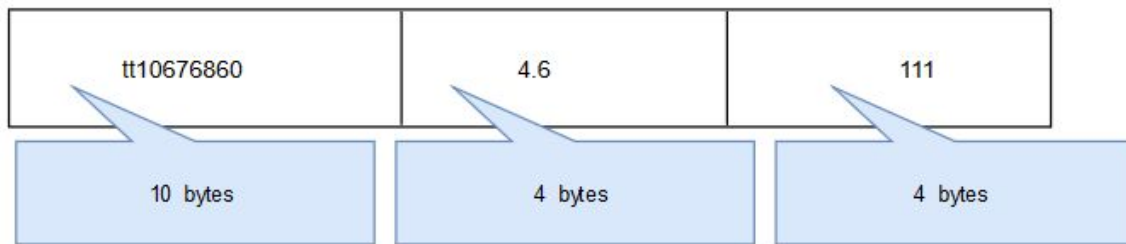
Storing each data item as a field:

Each data item is represented with a fixed number of bytes and they are stored as a field. The following shows the sample of our data item.

- 1) The string attribute "tconst" -> "tt10676860" has 10 characters and 10 bytes, since each character is considered as one byte.
- 2) The float attribute "averageRating" -> "4.6" requires only 1 decimal digit, hence 4 bytes is sufficient for this data type.
- 3) The integer attribute "numVotes" -> highest value is "2,279,223", which requires at least approximately 22 bits to represent the data type, hence 4 bytes will be the ideal data type.

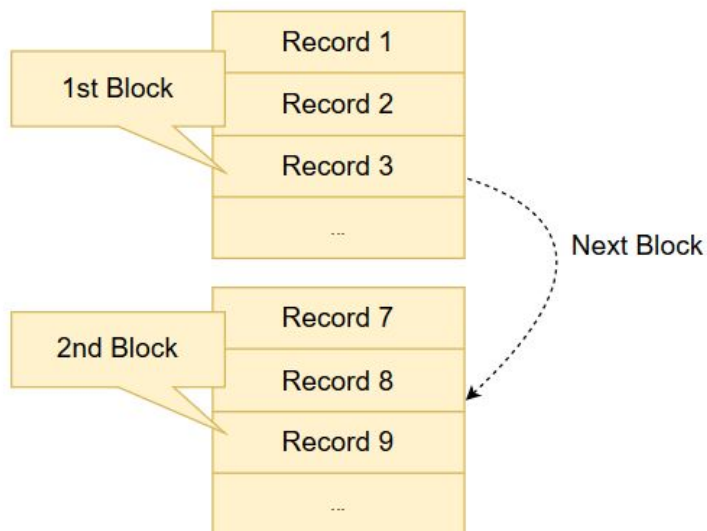
Packing fields into records:

A typical record with a fixed format with fixed length is as shown below:

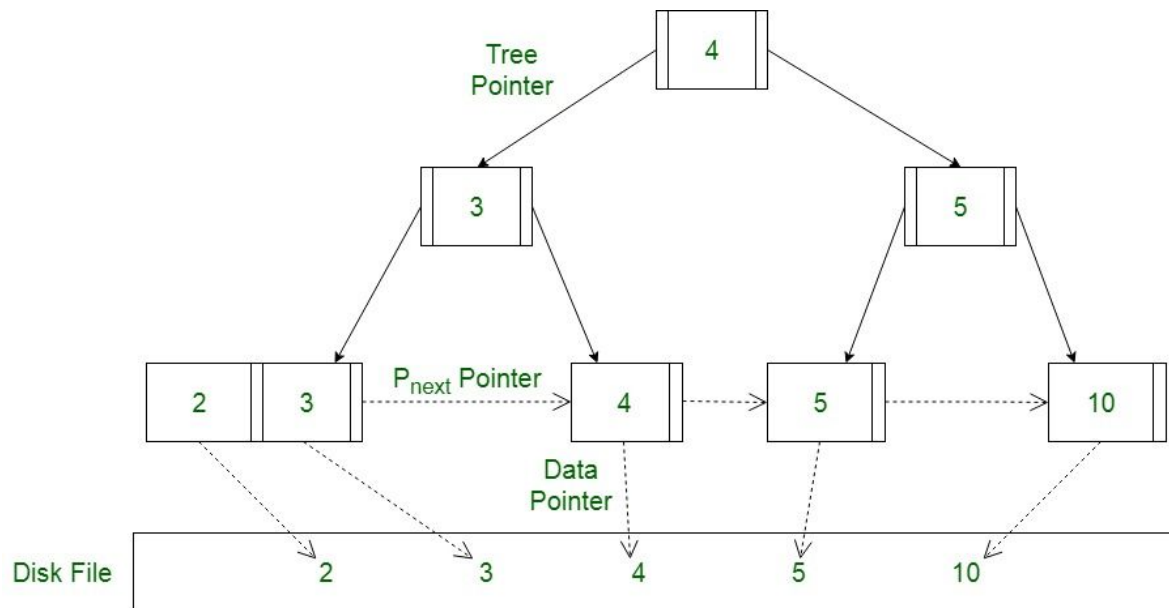


Packing records into blocks:

The records are unspanned records, which means records must be within one block. It is much simpler to manage the data in this way. The blocks are organized on disks sequentially since all blocks have a fixed length, separation is not needed. The current block is physically contiguous with the next block, so on and so forth.



## B+ Tree Overview



A B+ tree is a data structure often used in the implementation of database indexes. The B+ tree consists of a single node at the top also known as root node. The root node points to two or more blocks called child nodes which have further child nodes and so on. The B+-Tree is called a balanced tree because every path from the root node to a leaf node is the same length.

The B+ tree is similar to a B tree except:

- B+ trees don't store data pointers in interior nodes, they are ONLY stored in leaf nodes. This means that interior nodes can fit more keys on the block of memory.
- The leaf nodes of B+ trees are linked, so doing a linear scan of all keys will require just one pass through all the leaf nodes. This property can be utilized for efficient search as well since data is stored only in leafs.

The primary value of B+ Tree is that it offers significant value by providing efficient data retrieval in block-oriented storage applications like file systems and databases.

B+ tree satisfies the following conditions:

- An internal node (nodes that aren't the root or leaves) must have a number of children  $d$  such that  $\lceil m/2 \rceil \leq d \leq m$
- The root node may have at least 2 children and up to  $m$  children
- The number of keys  $k$  that an internal node may carry is  $\lceil m/2 \rceil - 1 \leq k \leq m - 1$
- Leaf nodes have no children, but the number of dictionary pairs  $k$  that a single leaf node may carry is  $\lceil m/2 \rceil - 1 \leq k \leq m - 1$

## Code Implementation

The B+ Tree program consists of the following:

1. **Key.java** - The Key class contains key value and the list of values (data pointer).

- *Attributes*

```
private float key; // Key Value  
List<Record> values; // Data Value: Array List of Record
```

- *Methods:*

- *get/set* methods
- *toString()* - prints the information of key

```
import java.util.ArrayList;

public class Key {

    private float key; // Key Value

    List<Record> values; // Data Value: Array List of Record

    // Initialize Key
    public Key(float key, Record value) {
        this.key = key;
        if (null == this.values) {
            values = new ArrayList<>();
        }
        this.values.add(value);
    }

    // Initialize Key
    public Key(float key) {
        this.key = key;
        this.values = new ArrayList<>();
    }

    // Return Key Value
    public float getKey() {
        return key;
    }

    // Set Key Value
    public void setKey(float key) {
        this.key = key;
    }

    // Get Key Data Values
    public List<Record> getValues() {
        return values;
    }

    // Set Key Data Values
    public void setValues(List<Record> values) {
        this.values = values;
    }

    // Print Key Information
    public String toString() {
        //return "<KEY>[Key= " + key + ", DataPointer= " + values + " ] ";
        return "<KEY>[Key= " + key + " ] ";
    }
}
```

2. **Node.java** - The Node class contains the ArrayList of Key class (Key.java) and ArrayList of Node Class which is used as a pointer to children.

- **Attributes:**

```
public class Node {
    private List<Key> keys; //Pointer to Keys

    private List<Node> children; //Pointer to Children

    private Node prev; //Pointer to Previous Node

    private Node next; //Pointer to Next Node

    private Node parent; //Pointer to Parent Node

    public boolean isLeaf; //Indicate if it's a leaf node
}
```

- **Methods:**

- get/set methods
- toString() - prints the information of node keys

```
// Initialize Node
public Node() {
    this.keys = new ArrayList<>();
    this.children = new ArrayList<>();
    this.prev = null;
    this.next = null;
}

// Return Array List of Keys
public List<Key> getKeys() {
    return keys;
}

// Set Keys of the Node
public void setKeys(List<Key> keys) {
    Iterator<Key> iter = keys.iterator();
    while (iter.hasNext()) {
        this.keys.add(iter.next());
    }
}

// Get List of Children Node
public List<Node> getChildren() {
    return children;
}

// Set Children of Node
public void setChildren(List<Node> children) {
    this.children = children;
}

// Get Next Node
public Node getNext() {
    return next;
}

// Set Next Node
public void setNext(Node next) {
    this.next = next;
}

// Get Parent Node
public Node getParent() {
    return parent;
}

// Set Parent Node
public void setParent(Node parent) {
    this.parent = parent;
}

// Get Previous Node
public Node getPrev() {
    return prev;
}

// Set Previous Node
public void setPrev(Node prev) {
    this.prev = prev;
}

// Return Node Information
@Override
public String toString() {
    return "Keys: " + keys.toString();
}
```

3. **BPTree.java** - The BPTree class contains methods in implementing the B+ tree.

- **Attributes:**

```
// B+ Tree Info
private int m;
private Node root;
private int height = 0;

// For Experiment Purposes
private int numNodes = 0;
private int numDeleted = 0;
private int numMerged = 0;
private int indexNodesAccess = 0;
private int dataBlocksAccess = 0;
private int uniqueKeysCount = 1;
private int recordsCountInANode = 0;
private int recordsCountTotal = 0;
private int numOfNodes = 0;
```

- **Methods:**

- *get/set* methods
- *Insert, Search, Delete*
- *merge, split* - merge/split nodes
- *print* - prints the information of nodes and trees

```
public BPTree(int order) {
    this.m = order; // Set B+ Tree M value
    this.root = null; // Set Root Node to null on default
}

// 1: Insertion Functions
public void insertKey(float key, Record value) {
    // 1: Empty B+ Tree => Create New Root Node
    if (null == this.root) {
        Node newNode = new Node();
        newNode.getKeys().add(new Key(key, value));
        this.root = newNode;
        this.root.isLeaf = true;
        this.root.setParent(null); // Since the root has no parent, parent set to null
    } else if (this.root.getChildren().isEmpty() && this.root.getKeys().size() < (this.m)) {
        // 2: Node is Not Full
        this.root.isLeaf = false;
        insertExternalNode(key, value, this.root);
    } else {
        // 3: Normal insert
        Node curr = this.root;

        // Traverse to the last leaf node
        while (!curr.getChildren().isEmpty()) {
            curr = curr.getChildren().get(searchInternalNode(key, curr.getKeys()));
        }

        insertExternalNode(key, value, curr);

        // External node is full => Split node
        if (curr.getKeys().size() == this.m) {
            splitExternalNode(curr, this.m);
        }

        // Increase number of nodes value
        numNodes++;
    }
}

private void insertExternalNode(float key, Record value, Node node) {
    // Find index of key to be inserted
    int index = searchInternalNode(key, node.getKeys());

    if (index != 0 && node.getKeys().get(index - 1).getKey() == key) {
        // Add the new value to the list
        node.getKeys().get(index - 1).getValues().add(value);
    } else {
        // Key is null or Add key and data
    }
}
```

(see full code in BPTree.java)



4. **Block.java** - The Block class adds records into blocks.

- *Attributes:*

```
public class Block {  
    private List<Record> records; //Pointer to Records
```

- *Methods:*

- *get/set methods*

```
public Block ()  
{  
    this.records = new ArrayList<>();  
}  
  
public List<Record> getRecords() {  
    return records;  
}  
  
public void setRecords(List<Record> records) {  
    Iterator<Record> iter = records.iterator();  
    while (iter.hasNext()) {  
        this.records.add(iter.next());  
    }  
}
```

5. **Record.java** - The Record class contains the records of the database.

- *Attributes:*

```
private String tconst;  
private float averageRating;  
private int numVotes;
```

- *Methods:*

- *get methods*

```
public Record(String tconst, float averageRating, int numVotes) {  
    this.tconst = tconst;  
    this.averageRating = averageRating;  
    this.numVotes = numVotes;  
}  
  
public String getTConst() {  
    return tconst;  
}  
  
public float getAverageRating() {  
    return averageRating;  
}  
  
public int getNumVotes() {  
    return numVotes;  
}
```

6. **Database.java** - The Database class contains methods to allocate/deallocate blocks, check database size and print records.

- **Attributes:**

```
public static int recordSize; // Record Size

private int poolSize; // Memory Pool Size
private int blockSize; // Block Size
private int freeSize; // Free Size
private int sizeUsed; // Size Used
private int allocated; // Allocated Size
private int remaining; // Number of Blocks Remaining
private int block; // Number of Blocks
public int recordsPerBlock; // Number of Records Per Block
public int totalNoOfRecords; // Total Number of Records
public int totalBlockSize; // Total Block Size
public int totalRecordSize; // Total Record Size
public int recordCounter = 0; // Number of Record
public int databaseSize = 0; // Database Size

private List<Block> listBlk; // Pointer to Data Block
private Block blk; // Block
```

- **Methods:**

- *get/set* methods
- *allocate/deallocate* - allocating/deallocating of blocks
- *print* - prints the information about the disk space size

```
public void printDatabaseInfo() {
    System.out.println("Memory Size: " + poolSize + " bytes");
    System.out.println("Block Size: " + blockSize + " bytes");
    System.out.println("Record Size: " + recordSize + " bytes");
    System.out.println("Free Size: " + freeSize + " bytes");
    System.out.println("Size Used: " + sizeUsed + " bytes");
    System.out.println("Available: " + remaining + " Blocks");
    System.out.println("Allocated: " + allocated + " Blocks");
    System.out.println("Total number of records: " + totalNoOfRecords);
    System.out.println("Number of records per block: " + recordsPerBlock);
    System.out.println("Number of Blocks: " + block);
    System.out.println("Total record size: " + totalRecordSize);
    System.out.println("Total block size: " + totalBlockSize);

    databaseSize = totalRecordSize + totalBlockSize;
    System.out.println("The Size of database: " + databaseSize);
}

public void setRecord(int totalNoOfRecords) {
    this.totalNoOfRecords = totalNoOfRecords;

    // round up to the nearest whole number to get minimum number of
    this.block = (int) Math.ceil((double) this.totalNoOfRecords / (double) recordSize);
}

public int getRecord() {
    return totalNoOfRecords;
}

public void printDataRecords() {
    System.out.println("List blk contents size: " + listBlk.size());

    for (int i = 0; i < listBlk.size(); i++) {
        // System.out.println("Each blk contents size: " +
        // listBlk.get(i).getRecords().size());

        for (int j = 0; j < listBlk.get(i).getRecords().size(); j++) {
            /*
             * System.out.println("Blk record tconst: " +
             * listBlk.get(i).getRecords().get(j).getTConst());
             * System.out.println("Blk record size: " +
             * listBlk.get(i).getRecords().get(j).getRecordSize());
             */
        }
    }
}
```

(see full code in Database.java)

7. **Main.java** - The main class reads the data.tsv file from the folder and creates the B+ Tree in the program. As it reads from the file, it creates a node and inserts into the tree. While making sure it follows the rules of B+ Tree, it will split or merge other nodes while it inserts in each line of the file. The main class outputs the experiment's details.

- *Attributes:*

```
// 1: Load Data File
String workingDir = System.getProperty("user.dir");
// String fileName = workingDir + "\\\" + "Project-BPlusT
// + "\\\" + "data.tsv";
String fileName = workingDir + "\\\" + "data.tsv";
File inputFile = new File(fileName);

// 2: Set Database and B+ Tree Size
BPTree tree = null;
Database db = null;

boolean exit1 = false;
boolean selected = false;
```

- *Methods:*

- *Scanner* - reads the data file
- Creates B+ tree
- *Experiment options* - carry out experiments

```
db.setRecord(recordCounter);
sc.close();
System.out.println("[Done Loading]");

// 4: Main Menu
boolean exit = false;
do {

    System.out.println("=====Experiments=====");
    System.out.println("1: Database Info");
    System.out.println("2: B+ Tree Info");
    System.out.println("3: Search Key with average rating of 8");
    System.out.println("4: Search Key Range with average rating of 7 to 9");
    System.out.println("5: Delete Key with average rating of 7");
    System.out.println("6: Quit");
    Scanner scan = new Scanner(System.in);
    int choice = scan.nextInt();

    switch (choice) {
        case 1:
            // Experiment 1
            System.out.println("=====Experiment 1=====");

            db.printDatabaseInfo();
            db.printDataRecords();
            break;
        case 2:
            // Experiment 2
            System.out.println("=====Experiment 2=====");
            tree.displayTreeInfo();
            tree.displayHeightInfo();
            System.out.print("\n");
            break;
        case 3:
            // Experiment 3
            System.out.println("=====Experiment 3=====");
            List<Record> searchValues = tree.searchKey(8);

            System.out.println("List of tconst: ");
            for (int j = 0; j < searchValues.size(); j++) {

                System.out.print(searchValues.get(j).getTConst() + " ");

                if (j % 100 == 0 && j != 0) {
                    System.out.print("\n");
                }
            }
        case 4:
            // Experiment 4
            System.out.println("=====Experiment 4=====");
            tree.deleteKey(7);
            break;
        case 5:
            // Experiment 5
            System.out.println("=====Experiment 5=====");
            tree.deleteKeyRange(7, 9);
            break;
        case 6:
            // Experiment 6
            System.out.println("=====Experiment 6=====");
            System.out.println("Exiting...");
            exit = true;
            break;
    }
} while (!exit);
```

(see full code in main.java)

# Experiments

## Experiment 1

Store the data (which is about IMDb movies and described in Part 4) on the disk and report the following statistics:

### **a) The number of blocks:**

#### **1- Block Size = 100 bytes**

Memory Size: 500000000 bytes (5mb)

Available :  $500000000/100 = 5000000$  blocks

Record Size: 18 bytes (tconst 10bytes(string), averageRating 4bytes(float), numVotes 4bytes(int))

Total number of records: 1070318

Number of record per block :  $100(\text{block size}) / 18(\text{record size}) = 5$  records(rounded down)

Number of blocks:  $1070318/5 = 214064$  blocks(rounded up)

Total record size: 19265724 bytes

Total block size:  $214064 * 100 = 21406400$  bytes

Remaining:  $5000000 - 214064 = 4785936$  blocks

#### **2- Block Size = 500 bytes**

Memory Size: 500000000 bytes (5mb)

Available:  $500000000/500 = 1000000$

Record Size: 18 bytes (tconst 10bytes(string), averageRating 4bytes(float), numVotes 4bytes(int))

Total number of records: 1070318

Number of record per block :  $500(\text{block size}) / 18(\text{record size}) = 27$  (rounded down)

Number of blocks:  $1070318/27 = 39642$  (rounded up)

Total record size: 19265724 bytes

Total block size:  $39642 * 500 = 19821000$  bytes

Remaining:  $1000000 - 39642 = 960358$  blocks

### **b) The size of database:**

The size of database = sum of the size of the relational data (total record size) + sum of the size of index nodes of the B Plus tree (total index nodes \* block size)

#### **1- Block Size = 100 bytes**

Size of database:  $19265724 + 25 * 100 = 19268224$

#### **2- Block Size = 500 bytes**

Size of database:  $19265724 + 5 * 500 = 19268224$

## Experiment 2

Build a B+ tree on the attribute "averageRating" by inserting the records sequentially and report the following statistics:

Each node of a B+ tree is stored as a block. An order for a B+ tree is the maximum branching factor which is defined as the maximum number of children a node may have. In this case, the maximum number of children corresponds to the maximum number of pointers in a node.

### **a) The parameter n of the B+ tree**

Each key in a B+ tree node is a float which occupies 4 bytes.

As the program is running Java HotSpot(TM) 64-Bit Server Virtual machine (VM), each pointer (to a node of a B+ tree or to a data block) occupies 8 bytes.

#### **1- Block Size = 100 bytes**

Let k be the number of keys

$$4k + 8 * (k + 1) \leq 100$$

$$k \leq 7.67$$

$$k = 7$$

Hence, the maximum number of keys is 7 and the maximum number of pointers is 7+1=8 that could be stored in a node of the B+ tree.

#### **2- Block Size = 500 bytes**

Let k be the number of keys

$$4k + 8 * (k + 1) \leq 500$$

$$k \leq 41$$

$$k = 41$$

Hence, the maximum number of keys is 41 and the maximum number of pointers is 41+1=42 that could be stored in a node of the B+ tree.

### **b) The number of nodes of the B+ tree**

#### **1- Block Size = 100 bytes**

	Nodes
Root (Level 1):	1
Internal (Level 2):	4
Leaf (Level 3):	20

Total number of nodes of the B+ tree: 25

## **2- Block Size = 500 bytes**

	Nodes
Root (Level 1):	1
Leaf (Level 2):	4

Total number of nodes of the B+ tree: 5

### **c) The height of the B+ tree, i.e., the number of levels of the B+ tree**

The leaf nodes must have at least 1,070,318 record pointers (excluding sequential pointers). Sequential pointers are the last pointers of each leaf node, their last pointers point to the first pointers of the adjacently connected next leaf node. Dense index is used for the lowest level at leaf nodes, which signifies that 1 pointer points to 1 record.

There are 91 unique keys for the float attribute "averageRating". The data structure of our B+ tree is built in such a way that all of the duplicate keys of attribute "averageRating" are stored contiguously together in the leaf nodes as an array list.

## **1- Block Size = 100 bytes**

The maximum number of keys of an internal node: 7

The maximum number of keys of a leaf node: 7

Let k be the number of keys

Let n be the number of levels of the B+ tree

$$91 \leq k * (k + 1) ^ (n-1)$$

$$91 \leq 7 * (8) ^ (n-1)$$

$$13 \leq (8) ^ (n-1)$$

$$\log(13) \leq \log((8) ^ (n-1))$$

$$\log(13) \leq (n-1) * \log(8)$$

$$(n-1) \geq \log(13) / \log(8)$$

$$n \geq 2.233479906$$

n = 3, hence the B+ tree has 3 levels.

## **2- Block Size = 500 bytes**

The maximum number of keys of an internal node: 41

The maximum number of keys of a leaf node: 41

Let k be the number of keys

Let n be the number of levels of the B+ tree

```

91 <= k * (k + 1) ^ (n-1)
91 <= 41 * (42) ^ (n-1)
2.219512195 <= (42) ^ (n-1)
log(2.219512195) <= log((42) ^ (n-1))
log(2.219512195) <= (n-1) * log(42)
(n-1) >= log(2.219512195) / log(42)
n >= 1.213311373
n = 2, hence the B+ tree has 2 levels.

```

#### **d) The root node and its child nodes (actual content)**

**The format of the output in the root and internal node is:**

Key:(Total Number of Records);  
 Ex) 3.2:(Empty);

As the root and internal node only contains the keys and pointers, the number of records are empty. The semicolon (;) denotes a pointer.

**The format of the output in the leaf node is:**

Key:(Total Number of Records);  
 Ex) 9.6:(2833);

As the leaf node contains the keys, pointers and duplicate records, the number of records are shown clearly. The semicolon (;) denotes a pointer.

#### **1- Block Size = 100 bytes**

**Console Output:**

**Printing level 1 (Root)**

3.2:();5.4:();7.5:();||

**Printing level 2**

1.6:();2.0:();2.4:();2.8:();||3.6:();4.0:();4.4:();5.0:();||5.9:();6.6:();7.1:();||8.0:();8.4:();8.8:();9.2:();9.6:();||

**Printing level 3**

1.0:(946);1.1:(250);1.2:(294);1.3:(276);1.4:(322);1.5:(318);||1.6:(473);1.7:(397);1.8:(560);1.9:(457);||2.0:(726);2.1:(697);2.2:(862);2.3:(858);||2.4:(1108);2.5:(1039);2.6:(1185);2.7:(1111);||2.8:(1631);2.9:(1286);3.0:(1799);3.1:(1563);||3.2:(2087);3.3:(1877);3.4:(2353);3.5:(2392);||3.6:(2956);3.7:(2760);3.8:(3570);3.9:(2993);||4.0:(4463);4.1:(3623);4.2:(5180);4.3:(4595);||4.4:(5632);4.5:(5399);4.6:(7037);4.7:(6511);4.8:(8765);4.9:(7131);||5.0:(10461);5.1:(8891);5.2:(12277);5.3:(10910);||5.4:(13305);5.5:(12514);5.6:(15400);5.7:(14646);5.8:(19331);||5.9:(16077);6.0:(22363);6.1:(19410);6.2:(25968);6.3:(22788);6.4:(26847);6.5:(24969);||6.6:(29200);6.7:(26913);6.8:(34628);6.9:(28134);7.0:(36626);||7.1:(30461);7.2:(38190);7.3:(32150);7.4:(35764);||7.5:(32012);7.6:(36460);7.7:(32329);7.8:(36607);7.9:(28376);||8.0:(33429);8.1:(26115);8.2:(29087);8.3:(21064);||8.4:(20196);8.5:(16799);8.6:(16522);8.7:(13177);||8.8:(12902);8.9:(8572);9.0:(9057);9.1:(5637);||9.2:(6390);9.3:(3828);9.4:(3724);9.5:(2312);||9.6:(2833);9.7:(1651);9.8:(1885);9.9:(623);10.0:(3026);||



Total number of nodes in B+ tree is: 25  
Total number of records in B+ tree is: 1070318

## **2- Block Size = 500 bytes**

### **Console Output:**

#### **Printing level 1 (Root)**

3.4:();5.5:();7.6:();||

#### **Printing level 2**

1.0:(946);1.1:(250);1.2:(294);1.3:(276);1.4:(322);1.5:(318);1.6:(473);1.7:(397);1.8:(560);1.9:(457);  
2.0:(726);2.1:(697);2.2:(862);2.3:(858);2.4:(1108);2.5:(1039);2.6:(1185);2.7:(1111);2.8:(1631);2.  
9:(1286);3.0:(1799);3.1:(1563);3.2:(2087);3.3:(1877);||3.4:(2353);3.5:(2392);3.6:(2956);3.7:(276  
0);3.8:(3570);3.9:(2993);4.0:(4463);4.1:(3623);4.2:(5180);4.3:(4595);4.4:(5632);4.5:(5399);4.6:(7  
037);4.7:(6511);4.8:(8765);4.9:(7131);5.0:(10461);5.1:(8891);5.2:(12277);5.3:(10910);5.4:(1330  
5);||5.5:(12514);5.6:(15400);5.7:(14646);5.8:(19331);5.9:(16077);6.0:(22363);6.1:(19410);6.2:(25  
968);6.3:(22788);6.4:(26847);6.5:(24969);6.6:(29200);6.7:(26913);6.8:(34628);6.9:(28134);7.0:(  
36626);7.1:(30461);7.2:(38190);7.3:(32150);7.4:(35764);7.5:(32012);||7.6:(36460);7.7:(32329);7.  
8:(36607);7.9:(28376);8.0:(33429);8.1:(26115);8.2:(29087);8.3:(21064);8.4:(20196);8.5:(16799);  
8.6:(16522);8.7:(13177);8.8:(12902);8.9:(8572);9.0:(9057);9.1:(5637);9.2:(6390);9.3:(3828);9.4:(  
3724);9.5:(2312);9.6:(2833);9.7:(1651);9.8:(1885);9.9:(623);10.0:(3026);||

Total number of nodes in B+ tree is: 5  
Total number of records in B+ tree is: 1070318

## Experiment 3

Retrieve the attribute “tconst” of those movies with the “averageRating” equal to 8 and report the following statistics:

### **a) The number and the content of index nodes the process accesses**

#### **1- Block Size = 100 bytes**

The total number of Index Nodes Access: 3

The content are:

Index Node Access: Node= [<KEY>[Key= 3.2] , <KEY>[Key= 5.4] , <KEY>[Key= 7.5] ]

Index Node Access: Node= [<KEY>[Key= 8.0] , <KEY>[Key= 8.4] , <KEY>[Key= 8.8] , <KEY>[Key= 9.2] , <KEY>[Key= 9.6] ]

Index Node Access: Node= [<KEY>[Key= 8.0] , <KEY>[Key= 8.1] , <KEY>[Key= 8.2] , <KEY>[Key= 8.3] ]

#### **2- Block Size = 500 bytes**

The total number of Index Nodes Access: 2

The content are:

Index Node Access: Node= [<KEY>[Key= 3.4] , <KEY>[Key= 5.5] , <KEY>[Key= 7.6] ]

Index Node Access: Node= [<KEY>[Key= 7.6] , <KEY>[Key= 7.7] , <KEY>[Key= 7.8] , <KEY>[Key= 7.9] , <KEY>[Key= 8.0] , <KEY>[Key= 8.1] , <KEY>[Key= 8.2] , <KEY>[Key= 8.3] , <KEY>[Key= 8.4] , <KEY>[Key= 8.5] , <KEY>[Key= 8.6] , <KEY>[Key= 8.7] , <KEY>[Key= 8.8] , <KEY>[Key= 8.9] , <KEY>[Key= 9.0] , <KEY>[Key= 9.1] , <KEY>[Key= 9.2] , <KEY>[Key= 9.3] , <KEY>[Key= 9.4] , <KEY>[Key= 9.5] , <KEY>[Key= 9.6] , <KEY>[Key= 9.7] , <KEY>[Key= 9.8] , <KEY>[Key= 9.9] , <KEY>[Key= 10.0] ]

**b) The number and the content of data blocks the process accesses**

**1- Block Size = 100 bytes**

The total number of Data Blocks Access: 1

The content are:

Data Block Access: Key=8.0

Value Size=33429 Records

Value (0)=Record@6615435c

**2- Block Size = 500 bytes**

The total number of Data Blocks Access: 1

The content are:

Data Block Access: Key=8.0

Value Size=33429 Records

Value (0)=Record@1936f0f5

The total number of records that contain averageRating equal to 8 is 33429 records.

654484584	16444736	65445402	65446238	65446433	65447437	65447495	65452628	65456794	65458091	65459472	65459598	65460938	65461859	65462678	65464426	65464454	65464556	65464578	t6546
65460854	65522330	65523772	65523780	65523978	65524108	65524438	65525724	65526612	65527499	65527914	65529596	65529606	65529704	65530072	65531834	65532028	65532194	65534018	t6547
65592402	65592688	65592986	65593898	65594196	65593934	65593938	65593972	65593982	65593988	65593998	65594000	65594004	65594034	65594088	65594946	65595054	65596972	65596958	t6548
65666342	65666966	65667638	65667740	65669424	65667150	65667408	65667120	65667428	65667138	65667340	65667228	65677042	65668094	65668698	65668148	65668158	65668356	65668358	t6549
65681154	65681408	65681700	65681740	65682408	65682450	65682454	65682402	65682430	65682432	65682436	65682434	65682438	65682492	65682496	65682506	65682510	65682512	65682512	t6550
65680014	65680098	65680156	65680214	65680218	65680516	65680526	65680538	65680528	65680528	65680648	65680872	65680912	656810126	65681042	65681124	65681300	65681480	65681596	t6551
65680634	65680684	65681592	65681762	65682708	65682672	65682676	65683060	65683198	65683256	65683608	65682662	65683702	65683704	65683746	65683748	65683912	65683928	65683928	t6552
65926020	65927650	65929336	65932966	65932608	65932748	65936038	65936950	65937134	65937718	65940784	65943632	65946354	65946710	65947326	65947562	65947994	65948806	65949508	t6553
65700130	65700126	657001810	65700194	65700207	65700236	65700364	657004810	65700514	65700511	65700818	657009978	657009990	657010686	657011370	657012058	65701266	65701280	65701354	t6554
65701154	65701690	65701674	65701658	65701680	65701717	65701768	657018010	657018156	65701814	65701948	657019564	657019564	657015492	6570155206	6570155448	657015640	657015732	65701576	t6555
65714784	65712290	657123032	657124352	65712366	657121608	657121604	657127058	657128796	657121890	657121902	657121930	657122130	657122186	6571222068	6571222526	6571222722	6571222696	6571222972	t6556
65728974	657284192	657284824	657285440	65728704	657289014	65728934	65728926	657289418	657289770	657290294	657290518	657291196	657291518	657292292	657292342	657293062	657293282	657296864	t6557
65736074	657370378	657370852	657371730	657372550	657373734	65737534	657375404	657378052	657378078	657378274	657379324	657379326	657380004	657380350	657380738	657381078	657381290	657381686	t6558
65740114	65741370	65741370	65741370	65741370	65741370	65741370	65741370	65741370	65741370	65741370	65741370	65741370	65741370	65741370	65741370	65741370	65741370	65741370	t6559
65752104	657527630	657529088	657529470	657529578	657530360	657530988	657531100	657531432	657531452	657533368	657534288	657534558	657535902	657535732	657535764	657536794	657536978	657539038	t6560
65760074	65760372	657604032	657604126	657604350	657606384	657608092	657												

16443854	16444736	16445462	16446238	16446430	16447372	16447998	16452628	16457964	16458010	16459472	16459968	16460938	16461850	16462678	16464426	16464466	16464544	16464566	16464578
16464580	16464592	16464604	16464616	16464628	16464640	16464652	16464664	16464676	16464688	16464700	16464712	16464724	16464736	16464748	16464760	16464772	16464784	16464796	16464808
16464820	16464832	16464844	16464856	16464868	16464880	16464892	16464904	16464916	16464928	16464940	16464952	16464964	16464976	16464988	16465000	16465012	16465024	16465036	16465048
16465080	16465092	16465104	16465116	16465128	16465140	16465152	16465164	16465176	16465188	16465200	16465212	16465224	16465236	16465248	16465260	16465272	16465284	16465296	16465308
16465340	16465352	16465364	16465376	16465388	16465400	16465412	16465424	16465436	16465448	16465460	16465472	16465484	16465496	16465508	16465520	16465532	16465544	16465556	16465568
16465600	16465612	16465624	16465636	16465648	16465660	16465672	16465684	16465696	16465708	16465720	16465732	16465744	16465756	16465768	16465780	16465792	16465804	16465816	16465828
16465860	16465872	16465884	16465896	16465908	16465920	16465932	16465944	16465956	16465968	16465980	16465992	16466004	16466016	16466028	16466040	16466052	16466064	16466076	16466088
16466120	16466132	16466144	16466156	16466168	16466180	16466192	16466204	16466216	16466228	16466240	16466252	16466264	16466276	16466288	16466300	16466312	16466324	16466336	16466348
16466400	16466412	16466424	16466436	16466448	16466460	16466472	16466484	16466496	16466508	16466520	16466532	16466544	16466556	16466568	16466580	16466592	16466604	16466616	16466628
16466680	16466692	16466704	16466716	16466728	16466740	16466752	16466764	16466776	16466788	16466800	16466812	16466824	16466836	16466848	16466860	16466872	16466884	16466896	16466908
16467000	16467012	16467024	16467036	16467048	16467060	16467072	16467084	16467096	16467108	16467120	16467132	16467144	16467156	16467168	16467180	16467192	16467204	16467216	16467228
16467280	16467292	16467304	16467316	16467328	16467340	16467352	16467364	16467376	16467388	16467400	16467412	16467424	16467436	16467448	16467460	16467472	16467484	16467496	16467508
16467560	16467572	16467584	16467596	16467608	16467620	16467632	16467644	16467656	16467668	16467680	16467692	16467704	16467716	16467728	16467740	16467752	16467764	16467776	16467788
16467840	16467852	16467864	16467876	16467888	16467900	16467912	16467924	16467936	16467948	16467960	16467972	16467984	16467996	16468008	16468020	16468032	16468044	16468056	16468068
16468120	16468132	16468144	16468156	16468168	16468180	16468192	16468204	16468216	16468228	16468240	16468252	16468264	16468276	16468288	16468300	16468312	16468324	16468336	16468348
16468400	16468412	16468424	16468436	16468448	16468460	16468472	16468484	16468496	16468508										

## Experiment 4

Retrieve the attribute “tconst” of those movies with the attribute “averageRating” from 7 to 9, both inclusively and report the following statistics:

### **a) The number and the content of index nodes the process accesses**

#### **1- Block Size = 100 bytes**

The total number of Index Nodes Access: 3

The content are:

Index Node Access: Node= [<KEY>[Key= 3.2] , <KEY>[Key= 5.4] , <KEY>[Key= 7.5] ]

Index Node Access: Node= [<KEY>[Key= 3.2] , <KEY>[Key= 5.4] , <KEY>[Key= 7.5] ]

Index Node Access: Node= [<KEY>[Key= 5.9] , <KEY>[Key= 6.6] , <KEY>[Key= 7.1] ]

#### **2- Block Size = 500 bytes**

The total number of Index Nodes Access: 2

The content are:

Index Node Access: Node= [<KEY>[Key= 3.4] , <KEY>[Key= 5.5] , <KEY>[Key= 7.6] ]

Index Node Access: Node= [<KEY>[Key= 3.4] , <KEY>[Key= 5.5] , <KEY>[Key= 7.6] ]



## **b) The number and the content of data blocks the process accesses**

### **1- Block Size = 100 bytes**

The total number of Data Block Access: 26

The content are:

```
Value (0)= Record@3cbbc1e0
Data Block Access: Key= 7.7
Value Size= 32329 Records
Value (0)= Record@35fb3008
Data Block Access: Key= 7.8
Value Size= 36607 Records
Value (0)= Record@7225790e
Data Block Access: Key= 7.9
Value Size= 28376 Records
Value (0)= Record@54a097cc
Data Block Access: Key= 8.0
Value Size= 33429 Records
Value (0)= Record@36f6e879
Data Block Access: Key= 8.1
Value Size= 26115 Records
Value (0)= Record@5a61f5df
Data Block Access: Key= 8.2
Value Size= 29087 Records
Value (0)= Record@3551a94
Data Block Access: Key= 8.3
Value Size= 21064 Records
Value (0)= Record@531be3c5
Data Block Access: Key= 8.4
Value Size= 20196 Records
Value (0)= Record@52af6cff
Data Block Access: Key= 8.5
Value Size= 16799 Records
Value (0)= Record@735b478
Data Block Access: Key= 8.6
Value Size= 16522 Records
Value (0)= Record@2c9f9fb0
Data Block Access: Key= 8.7
Value Size= 13177 Records
Value (0)= Record@2096442d
Data Block Access: Key= 8.8
Value Size= 12902 Records
Value (0)= Record@9f70c54
Data Block Access: Key= 8.9
Value Size= 8572 Records
Value (0)= Record@234bef66
Data Block Access: Key= 9.0
Value Size= 9057 Records
Value (0)= Record@737996a0
Data Block Access: Key= 9.1
Value Size= 5637 Records
Value (0)= Record@61dc03ce
```

## **2- Block Size = 500 bytes**

The total number of Data Block Access: 46

The content are:

```
Value Size= 13177 Records
Value (0)= Record@97e1986
Data Block Access: Key= 8.8
Value Size= 12902 Records
Value (0)= Record@26f67b76
Data Block Access: Key= 8.9
Value Size= 8572 Records
Value (0)= Record@153f5a29
Data Block Access: Key= 9.0
Value Size= 9057 Records
Value (0)= Record@7f560810
Data Block Access: Key= 9.1
Value Size= 5637 Records
Value (0)= Record@69d9c55
Data Block Access: Key= 9.2
Value Size= 6390 Records
Value (0)= Record@13a57a3b
Data Block Access: Key= 9.3
Value Size= 3828 Records
Value (0)= Record@7ca48474
Data Block Access: Key= 9.4
Value Size= 3724 Records
Value (0)= Record@337d0578
Data Block Access: Key= 9.5
Value Size= 2312 Records
Value (0)= Record@59e84876
Data Block Access: Key= 9.6
Value Size= 2833 Records
Value (0)= Record@61a485d2
Data Block Access: Key= 9.7
Value Size= 1651 Records
Value (0)= Record@39fb3ab6
Data Block Access: Key= 9.8
Value Size= 1885 Records
Value (0)= Record@6276ae34
Data Block Access: Key= 9.9
Value Size= 623 Records
Value (0)= Record@7946e1f4
Data Block Access: Key= 10.0
Value Size= 3026 Records
Value (0)= Record@3c09711b
```



Total Records with the attribute “averageRating” from 7 to 9: 545,895

**1- Block Size = 100 bytes**

12651798	12663558	12663864	12664420	12666476	12666596	12666616	12667764	12667868	12668182	12669896	12671934	12675236	12676194	12677542	12678510	12679162	12679614	12679896	12680442	12680442
12854780	12857832	12860736	12860816	12862640	12862748	12872458	12873184	12876480	12876798	12882128	12883826	12886374	12889996	12891812	12897234	12898414	12899338	12902880	12906870	12908940
13020158	13091758	13091952	13094048	13094820	13099584	13099550	13099600	13097392	13099110	13101232	13101464	13103144	13104446	13106600	13107718	13109254	13109850	13110684	13122546	
13298854	13321444	13322448	13322456	13322464	13322286	13323286	13323286	13323286	13323286	13324268	13324268	13324268	13324268	13324268	13324268	13324268	13324268	13324268	13324268	
13439084	13439084	13439084	13439084	13439084	13439084	13439084	13439084	13439084	13439084	13439084	13439084	13439084	13439084	13439084	13439084	13439084	13439084	13439084	13439084	
13588252	13588794	13589180	13591480	13591992	13592534	13593484	13594804	13595614	13595518	13595826	13597358	13598490	13599088	13599424	13599844	13604212	13604972	13609250	13615910	
13735212	13735428	13735528	13735878	13738632	13739186	13741988	13744724	13745628	13745628	13751298	13752146	13753548	13755478	13755478	13755478	13755478	13755478	13755478	13755478	
14075958	14077052	14078152	14081300	14083134	14083166	14085460	14085460	14085460	14085460	14085460	14085460	14085460	14085460	14085460	14085460	14085460	14085460	14085460	14085460	
14208078	14208100	14208538	14207998	14208284	14208284	14208990	14208984	14210386	14210428	14216902	14215928	14216864	14216864	14217688	14217688	14218452	14218474	14218490	14219170	
14325130	14329274	14332066	14332782	14334940	14335210	14335596	14335598	14339288	14342002	14342718	14343722	14343794	14345214	14345244	14345488	14356092	14360200	14362622	14362986	
14495170	14501832	14503026	14504804	14504862	14512534	14512842	14513164	14513994	14513994	14516158	14520256	14520966	14521500	14523138	14524090	14525108	14534044	14531604	14531696	
14852326	14852272	14854532	14854536	14854854	14854858	14854858	14854858	14854858	14854858	14854858	14854858	14854858	14854858	14854858	14854858	14854858	14854858	14854858	14854858	
15051008	15055344	15056832	15057276	15057280	15057286	15057288	15057292	15058412	15058552	15060212	15073396	15074174	15075072	15075072	15075072	15075072	15082450	15082482	15083176	
15198548	15200632	15201254	15202928	15203540	15203726	15203822	15204724	15206488	15206488	15206880	15208888	15209448	15213992	15214122	15215602	15215782	15216416	15216522	15216552	
15306852	15309728	15310228	15313236	15314190	15315102	15315108	15315212	15326684	15327118	15343740	15353798	15341464	15342484	15342494	15342694	15344066	15344954	15345398	15345734	
15677142	15677148	15677158	15677172	15677174	15677192	15677194	15677210													

**2- Block Size = 500 bytes**

[illegible]



## Experiment 5

**a) The number of times that a node is deleted (or two nodes are merged) during the process of the updating the B+ tree**

### 1- Block Size = 100 bytes

The total number of deleted nodes is 0

The total number of merged nodes is 0

### 2- Block Size = 500 bytes

The total number of deleted nodes is 0

The total number of merged nodes is 0

**b) The height of the updated B+ tree**

### 1- Block Size = 100 bytes

The tree height is 3.

### 2- Block Size = 500 bytes

The tree height is 2.

**c) The root node and its child nodes of the updated B+ tree**

### 1- Block Size = 100 bytes

**Console Output:**

**Printing level 1 (Root)**

3.2:();5.4:();7.5:();||

**Printing level 2**

1.6:();2.0:();2.4:();2.8:();||3.6:();4.0:();4.4:();5.0:();||5.9:();6.6:();7.1:();||8.0:();8.4:();8.8:();9.2:();9.6:();||

**Printing level 3**

1.0:(946);1.1:(250);1.2:(294);1.3:(276);1.4:(322);1.5:(318);||1.6:(473);1.7:(397);1.8:(560);1.9:(457);||2.0:(726);2.1:(697);2.2:(862);2.3:(858);||2.4:(1108);2.5:(1039);2.6:(1185);2.7:(1111);||2.8:(1631);2.9:(1286);3.0:(1799);3.1:(1563);||3.2:(2087);3.3:(1877);3.4:(2353);3.5:(2392);||3.6:(2956);3.7:(2760);3.8:(3570);3.9:(2993);||4.0:(4463);4.1:(3623);4.2:(5180);4.3:(4595);||4.4:(5632);4.5:(5399);4.6:(7037);4.7:(6511);4.8:(8765);4.9:(7131);||5.0:(10461);5.1:(8891);5.2:(12277);5.3:(10910);||5.4:(13305);5.5:(12514);5.6:(15400);5.7:(14646);5.8:(19331);||5.9:(16077);6.0:(22363);6.1:(19410);6.2:(25968);6.3:(22788);6.4:(26847);6.5:(24969);||6.6:(29200);6.7:(26913);6.8:(34628);6.9:(28

134);||7.1:(30461);7.2:(38190);7.3:(32150);7.4:(35764);||7.5:(32012);7.6:(36460);7.7:(32329);7.8:  
:(36607);7.9:(28376);||8.0:(33429);8.1:(26115);8.2:(29087);8.3:(21064);||8.4:(20196);8.5:(16799)  
;8.6:(16522);8.7:(13177);||8.8:(12902);8.9:(8572);9.0:(9057);9.1:(5637);||9.2:(6390);9.3:(3828);9.  
4:(3724);9.5:(2312);||9.6:(2833);9.7:(1651);9.8:(1885);9.9:(623);10.0:(3026);||

Total number of nodes in B+ tree is: 25

Total number of records in B+ tree is: 1033692

## **2- Block Size = 500 bytes**

**Console Output:**

**Printing level 1 (Root)**

3.4:();5.5:();7.6:();||

**Printing level 2**

1.0:(946);1.1:(250);1.2:(294);1.3:(276);1.4:(322);1.5:(318);1.6:(473);1.7:(397);1.8:(560);1.9:(457)  
;2.0:(726);2.1:(697);2.2:(862);2.3:(858);2.4:(1108);2.5:(1039);2.6:(1185);2.7:(1111);2.8:(1631);2.  
9:(1286);3.0:(1799);3.1:(1563);3.2:(2087);3.3:(1877);||3.4:(2353);3.5:(2392);3.6:(2956);3.7:(276  
0);3.8:(3570);3.9:(2993);4.0:(4463);4.1:(3623);4.2:(5180);4.3:(4595);4.4:(5632);4.5:(5399);4.6:(7  
037);4.7:(6511);4.8:(8765);4.9:(7131);5.0:(10461);5.1:(8891);5.2:(12277);5.3:(10910);5.4:(1330  
5);||5.5:(12514);5.6:(15400);5.7:(14646);5.8:(19331);5.9:(16077);6.0:(22363);6.1:(19410);6.2:(25  
968);6.3:(22788);6.4:(26847);6.5:(24969);6.6:(29200);6.7:(26913);6.8:(34628);6.9:(28134);7.1:(  
30461);7.2:(38190);7.3:(32150);7.4:(35764);7.5:(32012);||7.6:(36460);7.7:(32329);7.8:(36607);7.  
9:(28376);8.0:(33429);8.1:(26115);8.2:(29087);8.3:(21064);8.4:(20196);8.5:(16799);8.6:(16522);  
8.7:(13177);8.8:(12902);8.9:(8572);9.0:(9057);9.1:(5637);9.2:(6390);9.3:(3828);9.4:(3724);9.5:(2  
312);9.6:(2833);9.7:(1651);9.8:(1885);9.9:(623);10.0:(3026);||

Total number of nodes in B+ tree is: 5

Total number of records in B+ tree is: 1033692