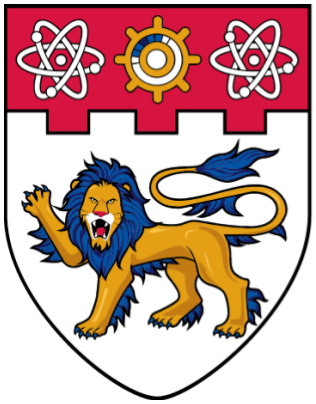


NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

CZ4031
Project 1 Report

Group 37

Goh Shan Ying (U1921497C)

Malcolm Tan (U2022160E)

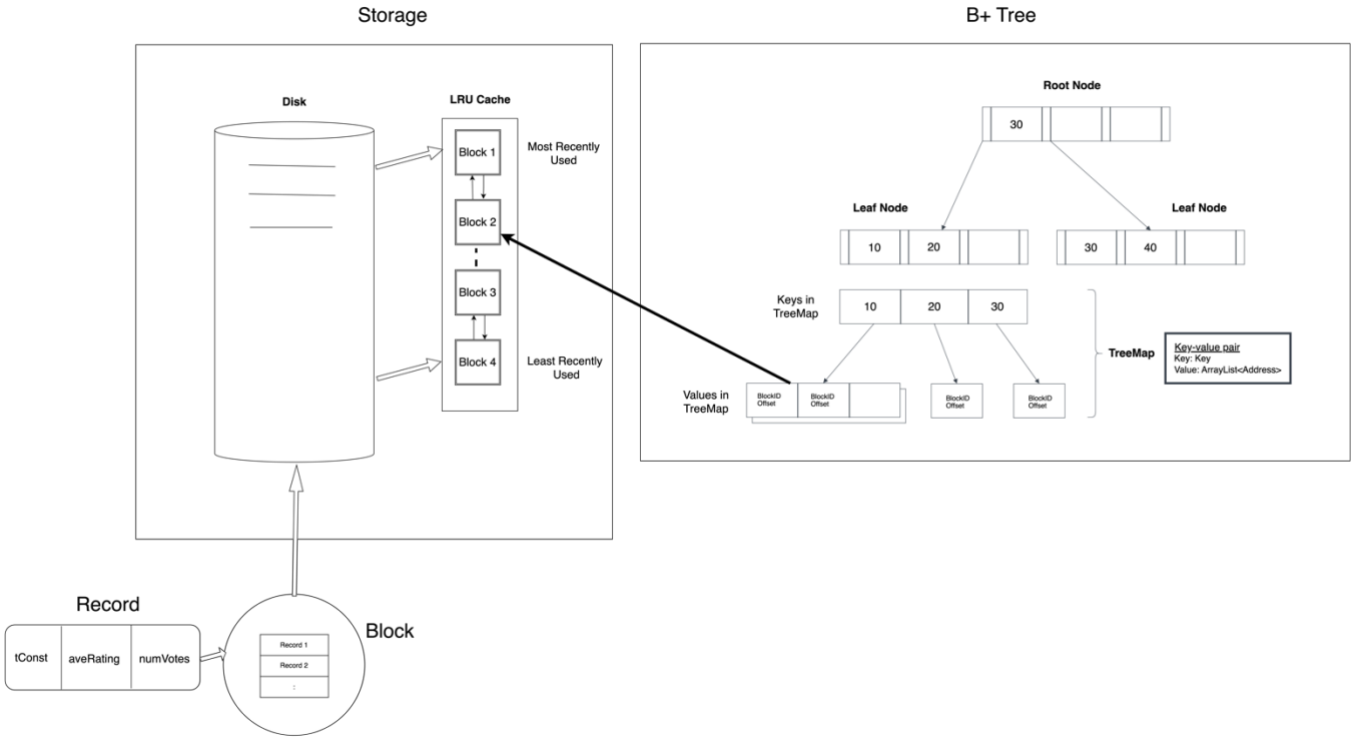
Ng Chi Hui (U1922243C)

Shannen Lee (U2021991G)

1. Introduction

This report outlines our design and implementation of a database management system, which includes the storage and indexing components. The underlying data structure of the indexing component is a B+ Tree and the entire system was developed using Java.

A. Project Overview



Our implementation consists of the following packages with the corresponding responsibilities:

- Utils (Parser): Parses the data from the ingested CSV and serializes the records to be stored into the database.
- BPTree: BP Tree package is responsible for implementing the B+ Tree data structure, which is commonly used for indexing and searching large datasets. The search, insertion, and deletion operations can be efficiently performed on a B+ Tree.
- Storage: Implementation of the database storage systems such as the Disk, Record, Address and Block components

B. Instructions

Clone the repository or download the zip file submitted. Instructions can be found in the Readme within the source code.

1. Running the project manually. Open the project up in your preferred IDE (I.e., IntelliJ/VSCode) and run the code.
2. Refer to Source Code Installation Guide for more details.

C. Contributions

Name	Contribution
Goh Shan Ying	1. Storage 2. B+ Tree Leaf and Non-Leaf Node Delete and Rebalancing
Malcom Tan	2. B+ Tree Indexing Insert 3. B+ Tree Leaf and Non-Leaf Node Delete and Rebalancing
Ng Chi Hui	1. Storage 2. B+ Tree Search 3. B+ Tree Leaf Node Delete
Shannen Lee	1. B+ Tree Indexing Insert 2. B+ Tree Leaf Node Delete and Rebalancing

2. Design of the Storage Component

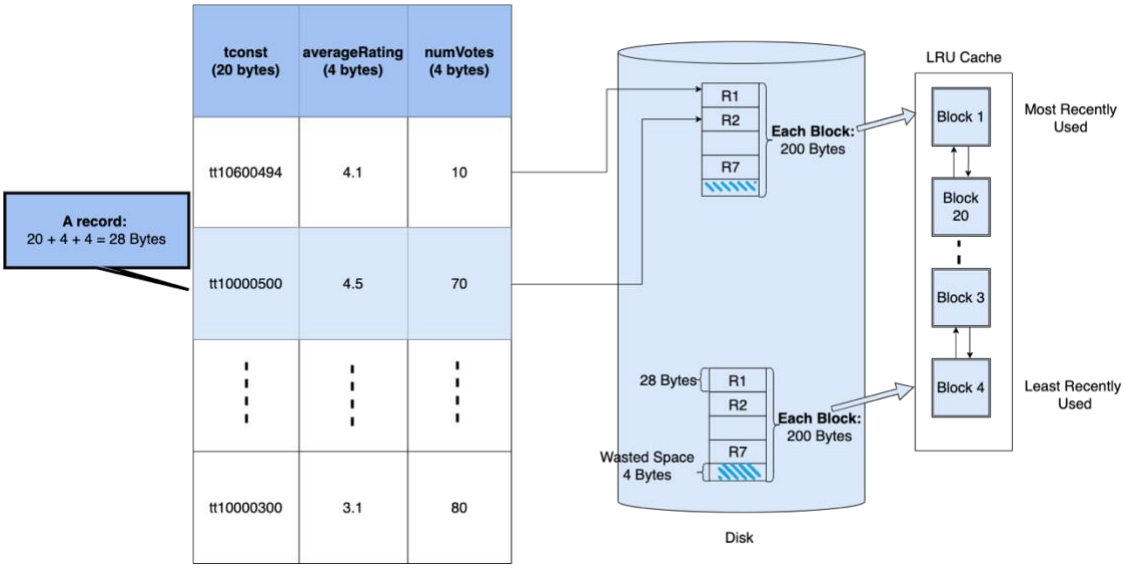


Figure 1: Overview of Database Storage Mechanisms

The internal of a file system instance constitutes the 4 main components, namely the Disk, the Block, the Record, and the Address.

A. Fields

As mentioned above, the data contains 3 types of attributes: tconst, averageRating and numVotes. These are the design considerations for the attributes:

tconst

Since the maximum size of the tconst string is 10 characters, it can be stored using an array of 10 characters. In Java, a character variable occupies 2 bytes of memory. As a result, using 10-character arrays for tconst would consume a total of 20 bytes.

averageRating

The averageRating attribute is a decimal number with a fixed precision of one decimal point, and its value ranges from 0.0 to 10.0. Therefore, the Float data type is a suitable option to store this attribute since it can store decimal numbers with a precision of 6 to 7 digits.
Using the Float data type to store the averageRating attribute would result in a size equivalent

to that of a float, which is 4 bytes.

numVotes

The numVotes attribute is a non-negative integer, the Integer (int) data type would be suitable to store this attribute. The int data type can handle values up to +2147483647, which is sufficient for the largest numVotes value in the dataset.

By using the Integer data type to store numVotes, the size of this field would be equivalent to that of an int, which is 4 bytes.

Field	Data Type	Size
tconst	String (string)	20 Bytes
averageRating	Float (float)	4 Bytes
numVotes	Integer (int)	4 Bytes

B. Records Serialization

There are two main types of field packing strategies that we can use namely fixed format with variable length (FFVL) and fixed format with fixed length (FFFL). Our group has chosen to implement the record serialization in the form of fixed format with fixed length strategy. Although FFFL implementation might cause some space to be wasted in the memory as compared to the FFVL strategy. However, if the data size were to grow large/exponentially, the space savings of using FFVL strategy might be significant and needs to be considered significantly.

There are a few main reasons for using FFFL instead of FFVL:

1. The project does not consider other data; thus, the format of the records remains consistent when reading in the datafile. After some analysis of the data, we found that the advantages of using FFVL storage for efficient storage are not significant in this case. For example, for records with tconst (a unique identifier for movie titles) that have a length between 9 and 10 characters, implementing FFVL would only save 1 byte of storage while requiring an additional byte to store the length information for variable length implementation. This means that no space is saved in the process.
2. FFVL tends to fare better with longer variable-length data, such as title or description fields, FFVL encoding can result in significant space savings. By using FFVL encoding, the length information is stored separately from the data itself, which allows for more efficient use of storage space. This is less applicable in our dataset.
3. Simpler Implementation

The end results of the serialization of each record can be seen in Figure 2. Each line of ingest from the TSV file will be serialized into a Record object with a total length of 28 Bytes.

Record (28 bytes)		
Tconst (20 bytes)	**averageRating (4 bytes)**	**numVotes (4 bytes)**

Figure 2: Data with a Record Object

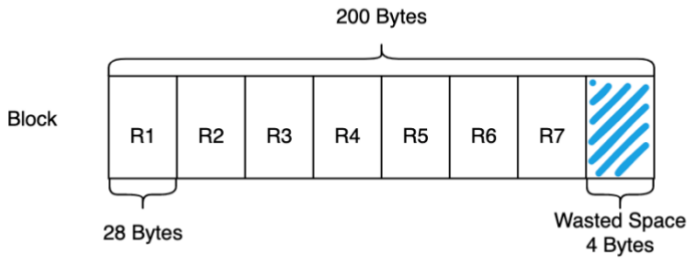
C. Block Storage

A Block class is built to implement a block in memory, and it packs records together. We have also chosen to implement the block to be unspanned, meaning that all records must be able to fit within a block. Although this will cause space to be wasted, it would allow easy searching for records within the blocks.

The records in the block are also stored in an unclustered manner meaning that they are not kept in any order. This is done so for the following reasons:

1. Increase the speed of insertion and deletion of the records.
2. Ease of Implementation of the unclustered index.

Considering that a block size is 200 bytes, and the size of a record is 28 bytes, each block would be able to hold 7 records. This can be visualized with the diagram below:



D. LRU Caching

LRU caching is used to improve the efficiency of accessing frequently used data and reduce the number of block accesses.

In an LRU cache, a fixed amount of memory is allocated to store data items. When a new item is added to the cache, it is placed at the front of a list that represents the cache. When the cache reaches its capacity limit, the item at the end of the list (the least recently used item) is removed to make space for the new item.

When a data item within a block is accessed from the cache, the block is moved to the front of the list to indicate that it has been used most recently. This way, the blocks that are used frequently are kept near the front of the list and are less likely to be removed from the cache. This reduces the number of block accesses from the disk when querying a record.

A typical implementation of LRUCache implementation is performed with double linked list and HashMap. However, to make use of the language feature that Java provides, a linkedHashMap implementation was able to solve the same problem. LinkedHashMap is a hash table and a linked list implementation of the Map interface, which means that it combines the fast access of a hash table with the ability to maintain the order of insertion of elements through a linked list.

E. Retrieval of Records

To stimulate block behavior in actual file systems in database, the Disk is designed to retrieve records by block. For record query, the program will return an Address<BlockId, Offset>. This allows us to determine the location of the block using the blockId where the record is stored on the disk.

- **Check if block in Cache:** Before initiating a record retrieval operation, the system first checks if the block containing the requested record is present in the LRU Cache.
- **Retrieve Block from Storage Medium:** If the block is found in the cache, the system can extract the requested record from the cached block. This reduces the number of block accesses from disk. If the requested block is not found in the cache, the system retrieves the entire block containing the record from the storage medium and stores it in the cache for future use.
- **Extract the record:** After the block is read into memory, the system searches for the specific record that needs to be retrieved using the offset in the block.

3. Design of the B+ Tree

A. Design of B+ Tree component

The B+ Tree is designed and implemented in “BplusTree.java”.

These are the classes we created and used:

1. Node

   Attributes:

   a. ArrayList<Integer> **keys** – to store the keys in each node. The reason behind choosing an ArrayList over a regular Java array is that it offers greater flexibility, such as allowing easier insertion and deletion of elements. This is because, with an ArrayList, we can simply call the add or remove methods, whereas with a regular array, we would need to manually shift elements around to accommodate the changes

   b. Node **rootNode** – the current rootNode

   c. NonLeafNode **parent** – the parent of the current node, will be null if the node does not have a parent

   d. **IsRoot** – boolean flag to check if the node is a root node. Returns true if is a root, false if it is not a root node

   e. **IsLeaf** – boolean flag to check if the node is a leaf node. Returns true if is a root, false if it is not a leaf node

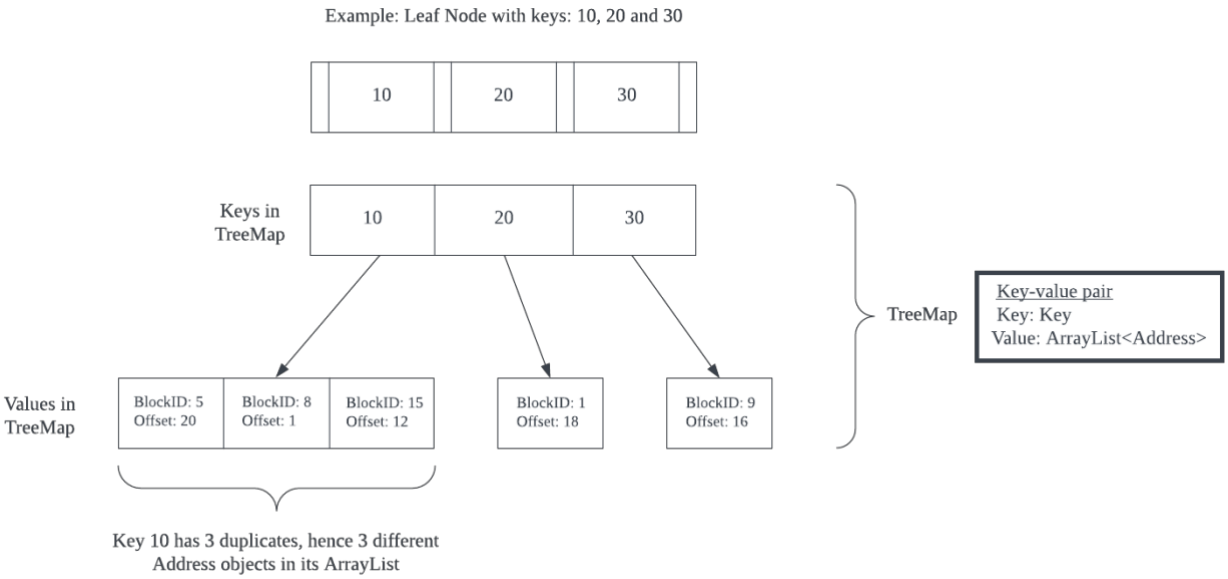
   f. **MinLeafNodeSize** – the minimum keys a leaf node must satisfy, which is calculated using the formula $\lfloor (n+1) \,/\, 2 \rfloor$, where n is the node size

   g. **MinNonLeafNodeSize** - the minimum keys a non-leaf node must satisfy, which is calculated using the formula $\lceil n \,/\, 2 \rceil$, where n is the node size

2. LeafNode (extends Node, has attributes of Node as well)

   Attributes:

   a. TreeMap <Integer, ArrayList<Address>> **map** – to store the keys of the records and the addresses corresponding to the key of the record (there will be multiple addresses for a same key due to duplicate records of the same key). The reason behind choosing a TreeMap is because TreeMap maintains the keys in sorted order, which is important for efficient searching, insertion, and deletion of records in the B+ tree. Additionally, TreeMap provides logarithmic time complexity for these operations, which is highly efficient for large data sets. Below is an example illustrating how we handle duplicate records with a TreeMap:



   b. ArrayList<Address> **records** – stores the address objects, which have getters and setters methods for blockID and offset. This information is used to get the address of the record in the disk

   c. LeafNode **nextNode –** the leaf node on the right-hand side of the current leaf node

   d. LeafNode **prevNode –** the leaf node on the left-hand side of the current leaf

node

3. NonLeafNode (extends Node, has attributes of Node as well)

   Attributes:

   a. ArrayList<Node> **children –** contains the children's nodes of the current node

B. Determining the Node size

When designing the B+ Tree, the maximum number of keys a node can contain is denoted by the parameter *n*. We can calculate the parameter *n* with the following formula:

$$n = \frac{block\ size - overhead}{pointer\ size + key\ size}, \textit{ where}$$

block size = 200 bytes
pointer size = 8 bytes (In the case of a 64-bit operating system)
overhead = 8 bytes
key size = 4 bytes (Integer data type for keys)

Following this formula, we can calculate that a B+ Tree with a block size of 200 bytes would maintain a maximum of 16 keys per node.

For block size = 200 bytes,

$$n = \frac{(200 - 8)}{8 + 4} = 16$$

C. Main functions of B+ tree

I. Search

a. Search Key Helper Function

Regardless of whether we are searching for a single value or values in a range, they will both use the searchKey() method. The searchKey() method is a helper method implemented to perform binary search on the nodes to find the position of the target key. The searchKey method takes four inputs, namely the left and right bounds of the search range, the target key, and a boolean flag indicating whether to search for the upper bound of the key or not. The method returns the index of the position where the key is found or the upper bound index of the key if it is not found. This method performs a standard binary search algorithm to search for the key and return the corresponding index.

b. Search Single Value

When performing a single value search, the searchValue() method takes in a node and a key as input. It then recursively descends through the tree until it finds a leaf node containing the key or determines that the key does not exist in the tree. At each point of the recursive search if the node taken in is a leaf node, it utilizes the searchKey function to find the position of the key that is in the current leaf node. Otherwise, it uses the searchKey function to locate the upper bound of the key and recursively descends to the child node at the corresponding position returned by the searchKey function.

The time complexity of searching a single value in a B+ tree is O (log n), where n is the number of nodes in the tree. This is because the search operation involves descending through the levels of the tree until the appropriate leaf node is found, and each level is searched using binary search, which has a time complexity of O (log m), where m is the number of keys in the node.

c. Search Values in a Range

The searchValuesInRange() method takes a minimum key, a maximum key, and a node as input, and returns all values in the tree whose keys fall within the specified range. This method also recursively descends through the tree. At each point of the recursive call, if the current node is leaf node, it utilizes the searchKey function to find the position of the minimum key in the node. Following which, it uses a while loop to advance to the next sibling node till the upper

range value of the search is reached. If the current node passed in is a non-leaf node, it uses the searchKey function to locate the upper bound of the key and recursively descends to the child node.

The time complexity of searching a range of values in a B+ tree is O (log n + k), where n is the number of nodes in the tree and k is the number of values in the range. This is because the search operation involves descending through the levels of the tree to find the leaf nodes that contain keys within the range, and then iterating over these nodes to retrieve the corresponding values. The number of leaf nodes that need to be visited is proportional to the size of the range, so the time complexity of the operation is O(k) in addition to the O (log n) time required for descending through the levels of the tree using binary search.

This makes B+ trees a very efficient data structure for performing range queries and other operations that involve searching for keys.

II. Insertion

The Insert function is the process of inserting a key to a B plus tree data structure while maintaining the B+ tree's properties. The insertion flow can be broken down into the following steps:

1. insertKey is first called, with parameters key and address passed as its arguments.
2. Inside insertKey, searchNode is called. searchNode takes in key as its argument and returns the leaf node where the key should be inserted to.
3. With the leaf node that is returned from searchNode, addRecord is called, which adds the key and address passed into the leaf node.
4. There are 4 different cases for addRecord:
    a. Current leaf node is empty
        i. Create new ArrayList and TreeMap for the current leaf node
        ii. Add key and address into the TreeMap
        iii. Add key into the ArrayList

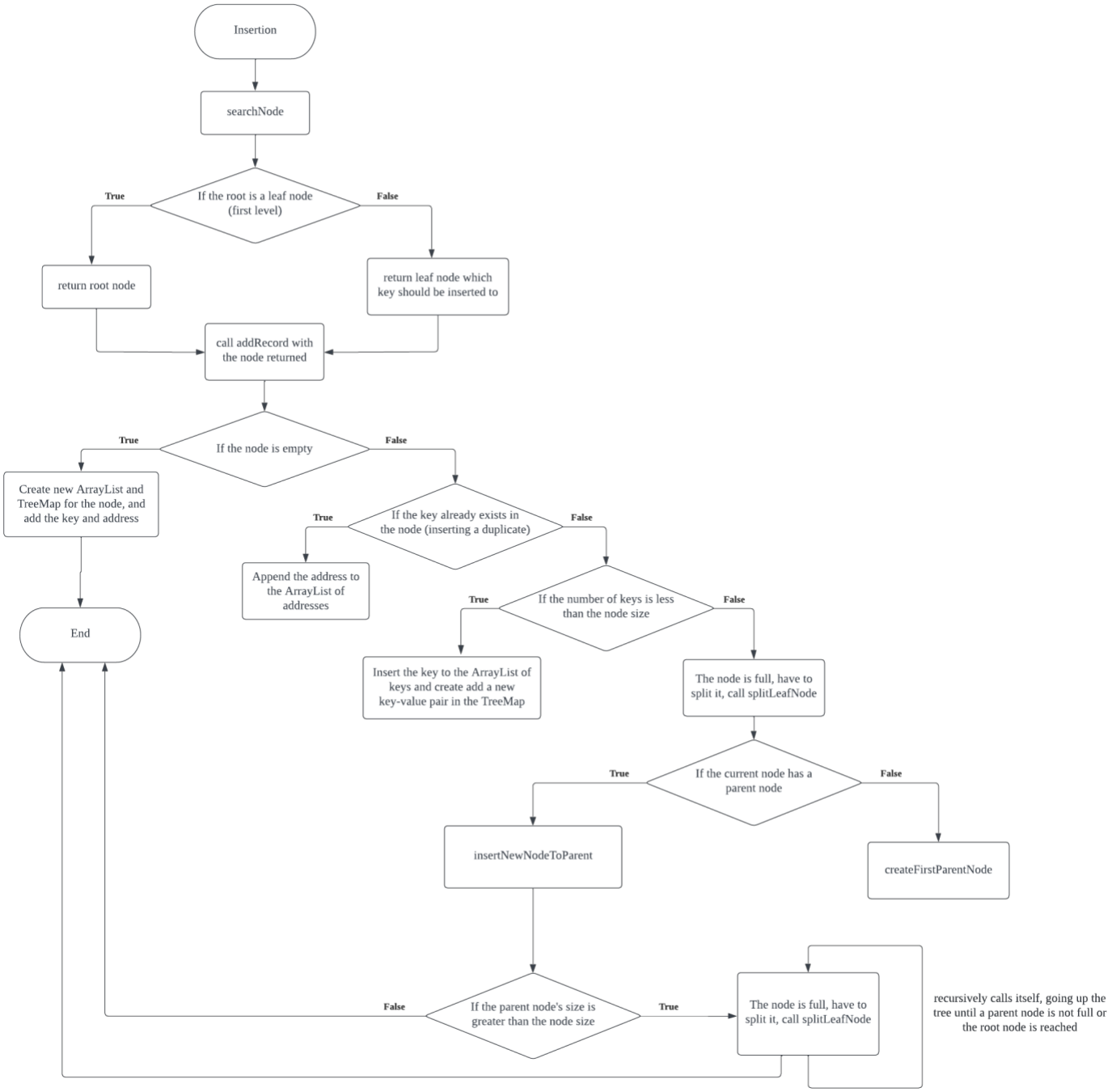
    b. Current leaf node already contains the key
        i. Get the existing value (ArrayList<Address> records) associated with key in the TreeMap (key-value pair)
        ii. Add address into records

    c. Current leaf node size is less than node size n
        i. Add key and address into the TreeMap
        ii. Add key into the ArrayList

    d. Current leaf node is full (size = n)
        i. Calls splitLeafNode, which splits the leaf node into newNode and the current node and inserts the newNode to the parent. However, if a parent is full, it calls a recursive function, splitNonLeafNode which repeatedly does the same

as splitLeafNode but on non-leaf node instead. It starts from the current node and goes back up until the root node, checking and modifying parent nodes that are full when returning up to the root node.

Flow Diagram
(Reference to Insertion_Flow_Diagram.png for clarity)



III. Deletion

The deleteKey function is the process of removing a key from a B plus tree data structure while maintaining the B+ tree's properties. The delete flow can be broken down into four steps:

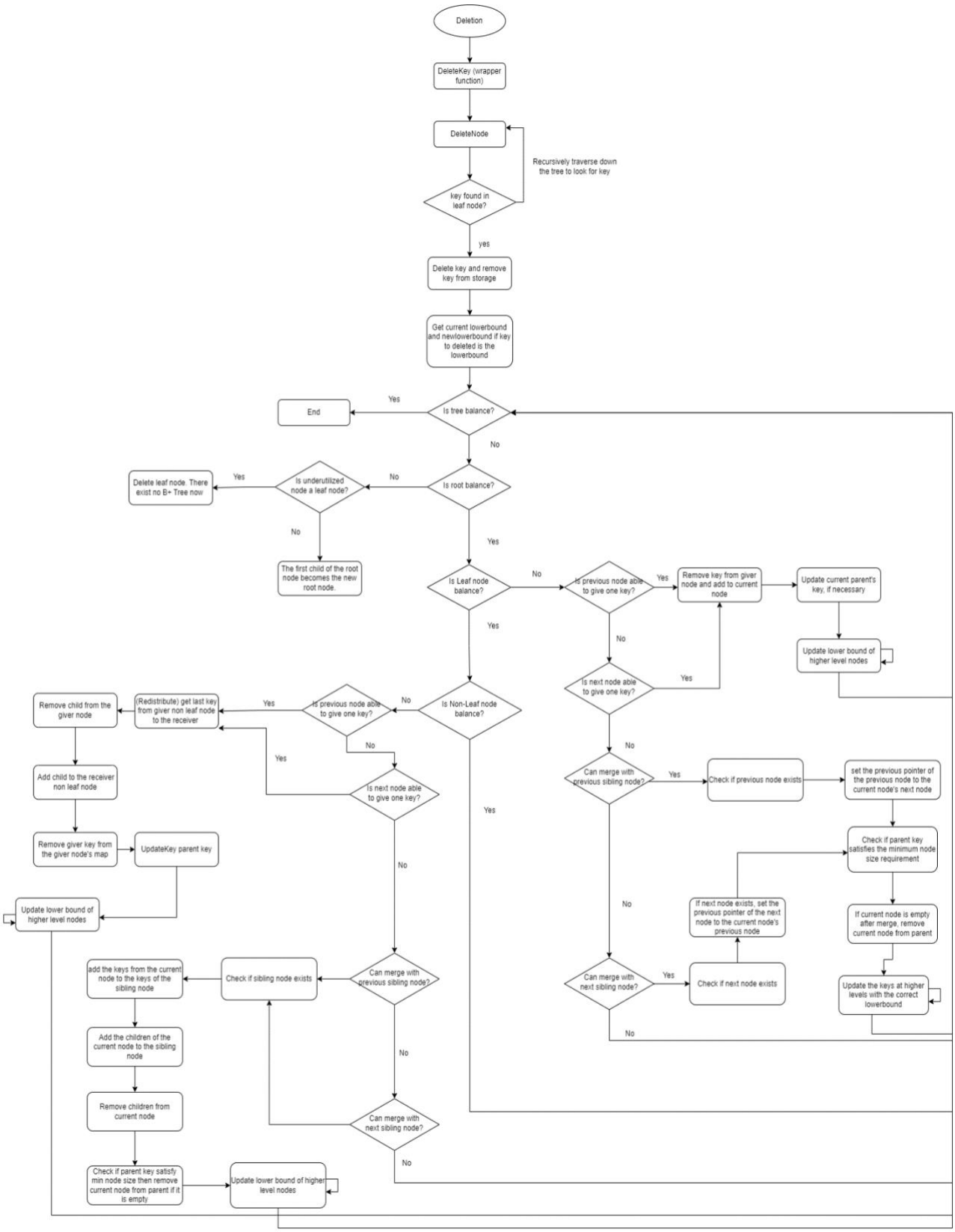
1. Look for key to delete recursively: To delete a key from a B plus tree, we need to traverse the tree recursively to find the key to delete.
    - If the key is found in a leaf node, we remove it from the node and remove the record from disk.
    - If the node becomes imbalanced after removing the key, we adjust the tree by merging or redistributing nodes.
    - If the key is not found, we return an error message.
2. Check if the deleted key is a lower bound and update the non-leaf nodes at higher levels: Before deleting the key from the leaf node, we check if the deleted key is a lower bound for that node. If the deleted key used to be the lower bound, we check if the leaf node can obtain a new lower bound by using the right key from the deleted key. If the leaf node can obtain a new lower bound, we update the tree recursively. If the leaf node cannot obtain a new lower bound, the lower bound remains the same.
3. Check if the B-tree is balanced recursively: The balance of a B plus tree is maintained by ensuring that each leaf node has a minimum of $\left\lfloor \frac{(n+1)}{2} \right\rfloor$ keys and each non leaf node has a minimum of $\left\lfloor \frac{n}{2} \right\rfloor$ keys, where n is the maximum number of keys in a node. If a node has too few or too many keys, it is considered unbalanced.
    - To address an unbalanced node, we first check if we can borrow a key from its sibling node. If we can, we perform a redistribution of keys. A leaf node's keys can be redistributed with a neighboring leaf node by moving a key from one node to the other. A non-leaf node's children can be redistributed with a neighboring non-leaf node by moving a child from one node to the other. We also update the parent node's keys accordingly.
    - If redistribution is not possible, we merge the nodes. Merging two leaf nodes involves combining their keys and children and adjusting the parent node accordingly. Merging two non-leaf nodes involves combining their keys and

children and adjusting the parent node accordingly. We also update the parent node's keys accordingly.

4. After merging or redistributing nodes, we need to update the parent node's keys to reflect the changes. We then need to update the tree recursively until all nodes are balanced.
    - Update the key at higher levels with the correct lower bound: After merging or redistributing nodes, we need to update the key at higher levels with the correct lower bound. This step ensures that the B-tree remains balanced and efficient.

Flow Diagram

(Reference to Deletion_Flow_Diagram.png for clarity)



5. Results

The following presents the results on disk sizes 200B and 500B.

A. 200B Disk Size

Experiment 1

Total No. Of Records stored:	1070318
Size of each Record:	28 Bytes
Number of Records stored in a block:	7
Number of Blocks allocated:	152903

Experiment 2

Parameter *n* of the B+ Tree:	16
Number of nodes in the B+ Tree:	1747
Height of the B+ Tree:	4

Contents of the root node:

Root Node									
1342	2631	3897	5970	8878	12134	20151	30034	49802	125979

Experiment 3

Number of index nodes accessed:	4
Number of data blocks accessed:	110
Average of “averageRating’s” values returned:	6.73
Runtime of retrieval process:	819500 nanoseconds
No. of Data Blocks accessed with Brute-Force:	152903
Runtime of Brute-Force Approach:	62530200 nanoseconds

Contents of first five index nodes:

Root Node									
1342	2631	3897	5970	8878	12134	20151	30034	49802	125979

First Child Node								
106	267	448	573	726	837	960	1122	1224

Second Child Node									
458	468	478	487	501	514	525	538	552	561

Third Child Node													
487	488	489	490	491	492	493	494	495	496	497	498	499	500

Experiment 4

Number of index nodes accessed:	85
Number of data blocks accessed:	942
Average of “averageRating’s” values returned:	6.73
Runtime of retrieval process:	428700 nanoseconds
No. of Data Blocks accessed with Brute-Force:	152903
Runtime of Brute-Force Approach:	235752100 nanoseconds

Contents of first five index nodes:

Root Node									
1342	2631	3897	5970	8878	12134	20151	30034	49802	125979

First Child Node							
20937	21812	22631	23444	24703	26119	27317	28769

Second Child Node									
28916	29004	29116	29268	29387	29594	29633	29743	29848	29959

Third Child Node								
29959	29962	29974	29975	29978	29982	29988	29996	30022

Fourth Child Node								
29959	29962	29974	29975	29978	29982	29988	29996	30022

Experiment 5

Number of deleted nodes:	0
Number of nodes in updated B+ tree:	1747
Height of updated B+ tree:	4
Runtime of retrieval process:	448600 nanoseconds
No. of Data Blocks accessed with Brute-Force:	152861
Runtime of Brute-Force Approach:	24102200 nanoseconds

Contents of the updated B+ tree root node:

Root Node									
1342	2631	3897	5970	8878	12134	20151	30034	49802	125979

B. 500B Disk Size

Experiment 1

Total No. Of Records stored:	1070318
Size of each Record:	28 Bytes
Number of Records stored in a block:	7
Number of Blocks allocated:	152903

Experiment 2

Parameter *n* of the B+ Tree:	16
Number of nodes in the B+ Tree:	1747
Height of the B+ Tree:	4

Contents of the root node:

Root Node									
1342	2631	3897	5970	8878	12134	20151	30034	49802	125979

Experiment 3

Number of index nodes accessed:	4
Number of data blocks accessed:	110
Average of “averageRating’s” values returned:	6.73
Runtime of retrieval process:	71150 nanoseconds
No. of Data Blocks accessed with Brute-Force:	152903
Runtime of Brute-Force Approach:	61292300 nanoseconds

Contents of the four index nodes:

Root Node									
1342	2631	3897	5970	8878	12134	20151	30034	49802	125979

First Child Node								
106	267	448	573	726	837	960	1122	1224

Second Child Node									
458	468	478	487	501	514	525	538	552	561

Third Child Node													
487	488	489	490	491	492	493	494	495	496	497	498	499	500

Experiment 4

Number of index nodes accessed:	85
Number of data blocks accessed:	942
Average of “averageRating’s” values returned:	6.73
Runtime of retrieval process:	565100 nanoseconds
No. of Data Blocks accessed with Brute-Force:	152903
Runtime of Brute-Force Approach	233722000 nanoseconds

Contents of first five index nodes:

Root Node									
1342	2631	3897	5970	8878	12134	20151	30034	49802	125979

First Child Node							
20937	21812	22631	23444	24703	26119	27317	28769

Second Child Node									
28916	29004	29116	29268	29387	29594	29633	29743	29848	29959

Third Child Node								
29959	29962	29974	29975	29978	29982	29988	29996	30022

Fourth Child Node								
29959	29962	29974	29975	29978	29982	29988	29996	30022

Experiment 5

Number of deleted nodes:	0
Number of nodes in updated B+ tree:	1747
Height of updated B+ tree:	4
Runtime of retrieval process:	319300 nanoseconds
No. of Data Blocks accessed with Brute-Force:	152861
Runtime of Brute-Force Approach	23603000 nanoseconds

Contents of the root node:

Root Node									
1342	2631	3897	5970	8878	12134	20151	30034	49802	125979

Source Code Installation Guide:

The codebase is stored under one cohesive package called Database-B-Plus Tree. Alternatively, you can access the project codebase via the GitHub link provided: https://github.com/Chihui8199/CZ4031-Project-1.git.

Here's how:

1. Open a web browser and go to https://github.com/Chihui8199/CZ4031-Project-1.git.
2. On the repository page, click on the green "Code" button on the right side of the screen.
3. Click on the "Download ZIP" option from the dropdown menu.
4. The zip file will begin to download. Once it is finished, you can unzip the folder and follow the steps provided to run the project.

Here are the steps to run the project from the zip file:

1) Unzip the folder and open the codebase (CZ4031 Project 1 Code) to an IDE of your choice.

2) In the unzipped folder, the /src folder is where the tsv data is stored. These files will be used to store the data in memory.

3) To run the code and see Experiments 1 to 5, open the main.java file in the IDE and run the file.

4) The output on the terminal / console window are the answers to experiments 1 till 5 along with descriptions of how memory allocation was done by our code