NANYANG TECHNOLOGICAL UNIVERSITY,

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

CZ4031 Database System Principles

Project 1

Group 41

Group Members:

Ong Kwang Wee (U2021723J)

Wong Jing Yen (U2020809H)

Kenny Tan Junrong (U1920056G)

Manimaran Manoj Kumar (U2021005E)

Alloysius Lim Zong Hong (U1922302F)

Contribution of Team Members

Task	Name
Storage Implementation	Ong Kwang Wee Kenny Tan Junrong
Base of B+ Tree Implementation	Wong Jing Yen
B+ Tree Implementation (Insertion)	Wong Jing Yen Kenny Tan Junrong Alloysius Lim Zong Hong Manimaran Manoj Kumar
B+ Tree Implementation (Search)	Wong Jing Yen Kenny Tan Junrong Manimaran Manoj Kumar
B+ Tree Implementation (Deletion)	Wong Jing Yen
Experiments	All members
Report	All members

Name	Contribution
Wong Jing Yen	20%
Kenny Tan Junrong	20%
Manimaran Manoj Kumar	20%
Ong Kwang Wee	20%
Alloysius Lim Zong Hong	20%

Introduction

Description

This current project focuses on the design and implementation of the two components of a database management system, mainly storage and indexing. It also demonstrates a B+ Tree implementation that supports inserting, searching and deleting operations.

Project Overview

The implementation consists of the following classes with the corresponding responsibilities:

- Main.java: The boundary class that accepts user input and invokes methods in other classes to conduct the experiments for the project
- BPlusTreeNode.java: Implementation of the B+ Tree structure through insertion and deletion of keys into nodes. It also supports searching of keys.
- Disk.java: Supports insertion and deletion of records into blocks. Also supports retrieval of data from specified records from the disk.
- Block.java: Implementation of Block structure that controls the storage of records
- Record.java: Implementation of Record structure that holds the fields (tconst, averageRating, and numVotes).
- Utils.java: To map unchanging values to respective variable constants that are used throughout the classes that aids to make the code more robust.

Program Run Instructions

This program is built and compiled to a .jar file type to ensure the ease of running the program. The steps below are to run the .jar file from Terminal or IDE.

Clone the repository from Github repository https://github.com/kennytaan/CZ4031-DBSP-Java or extract the zipfile submitted. Ensure that you have any supporting Java IDE (E.g. Microsoft Visual Studios, Eclipse, IntelliJ) which may require additional installations like JDKs to open the project folder. A recommended installation guide is provided on README.md on github repository.

These are the following steps to run the program:

1. Navigate to the Main.java.file on IDE or Terminal (PATH: /CZ4031-DBSP-Java)
2. Running Main.java through the IDE or Terminal
3. Experiments 1-5 will be carried out with disk block size of 200MB and respective outputs can be seen under the Terminal after it has completed running.

4. The terminal will prompt for an user input to:
    - [1] Repeat Experiments 1-5 with disk block size of 500MB
    - [2] Exit the program
5. The program will continue to run only if “1” is entered, else it will prompt for the correct user input or exit the program.
6. Experiments 1-5 will be carried out with disk block size of 500MB and respective outputs can be seen under the Terminal after it has completed running.

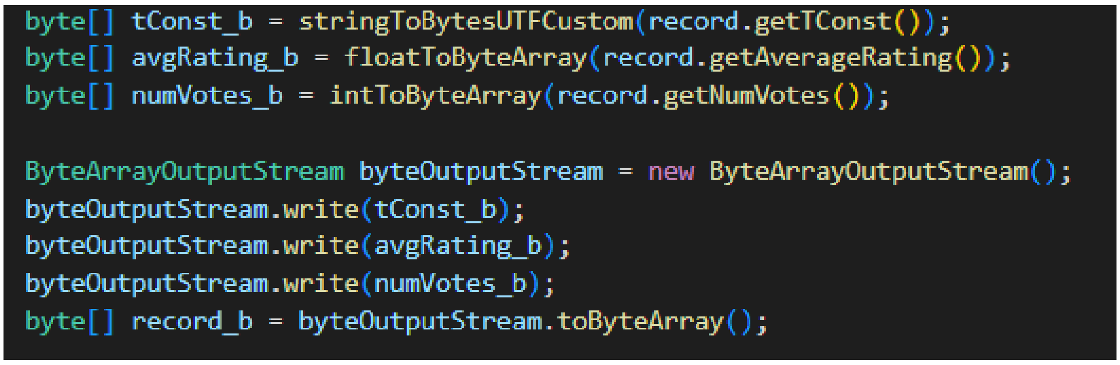
Design of Storage components

Records

The data.tsv file consists of 3 column values: tconst, averageRating, numVotes.
Each line read from the tsv file is split into fields which are then packed into a single record as follows:

- tconst -> e.g. tt9916778 -> char[] (10characters with 2bytes each = 20bytes)
- averageRating -> e.g. 9.9 -> float (4 bytes)
- numVotes -> e.g. 4582 -> int (4 bytes)

Total record size = 28 bytes

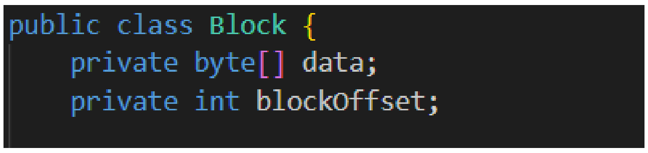


```
byte[] tConst_b = stringToBytesUTFCustom(record.getTConst());
byte[] avgRating_b = floatToByteArray(record.getAverageRating());
byte[] numVotes_b = intToByteArray(record.getNumVotes());

ByteArrayOutputStream byteOutputStream = new ByteArrayOutputStream();
byteOutputStream.write(tConst_b);
byteOutputStream.write(avgRating_b);
byteOutputStream.write(numVotes_b);
byte[] record_b = byteOutputStream.toByteArray();
```

Figure 1: Logic to handle conversion of datatype to Byte Array

As show in Fig 1, each record attribute is being converted to a byte array to ensure consistency in data size. Thus, no additional memory is required to indicate the size of each field.

Blocks



```
public class Block {
    private byte[] data;
    private int blockOffset;
```

Figure 2: Block class

The class Block consist of 2 private declarations:

- data
    - o Consist of the data bytes of a single record
- blockOffset
    - o Acts as a pointer to determine the next location of the next record. The offset will be adjusted if a new block is created.

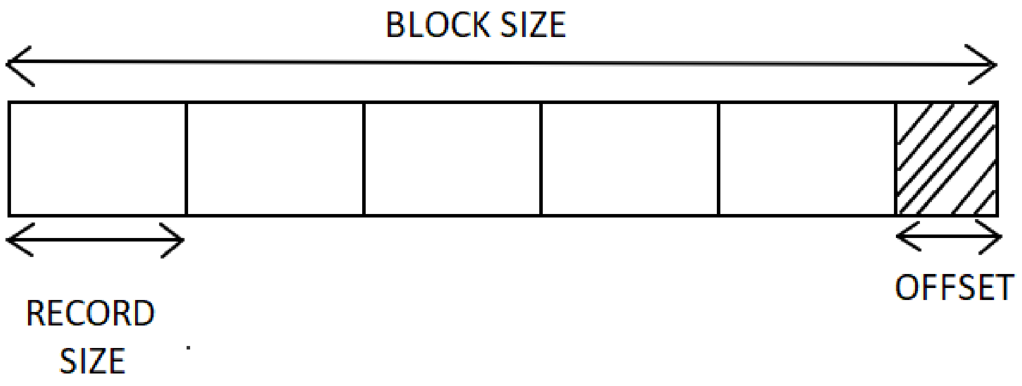
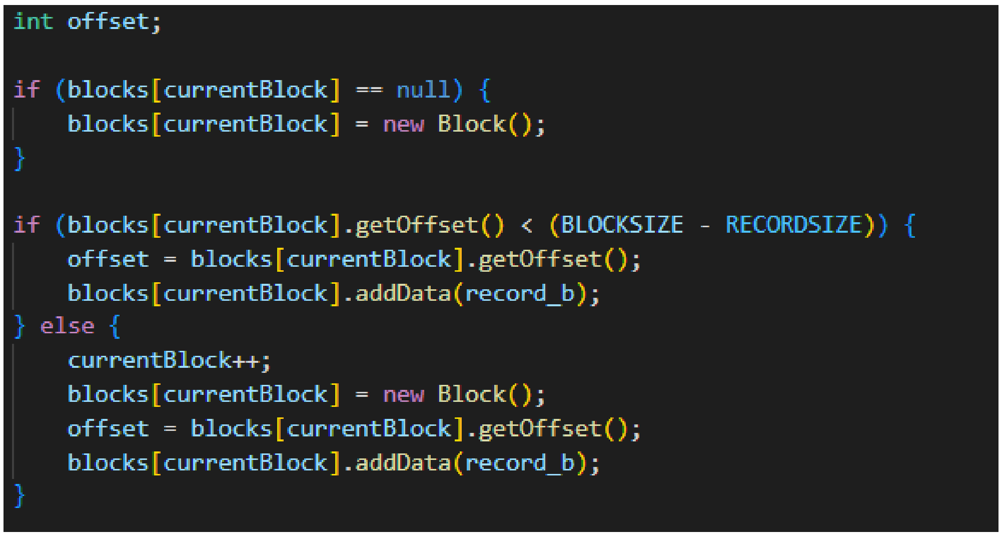


Figure 3: How records are packed into block

This project utilises the unspanned record method to pack records into blocks. This means that the record can only be stored inside a block if it can be completely stored into it. A record cannot be stored in more than one block. The reasons for choosing this method are as follows:

- Record size is smaller than block size.
- Access time of a record is much faster.
- It is a simpler way for storage implementation as compared to the spanned record method.



```
int offset;

if (blocks[currentBlock] == null) {
    blocks[currentBlock] = new Block();
}

if (blocks[currentBlock].getOffset() < (BLOCKSIZE - RECORDSIZE)) {
    offset = blocks[currentBlock].getOffset();
    blocks[currentBlock].addData(record_b);
} else {
    currentBlock++;
    blocks[currentBlock] = new Block();
    offset = blocks[currentBlock].getOffset();
    blocks[currentBlock].addData(record_b);
}
```

Figure 4: Logic to handle packing of records into a block

Fig 4 shows how we implement the part where the records can only be inserted into the block if its size is smaller than the offset. A new block will be created to store the record if the record's size is greater than the offset and cannot be inserted into the block.

Disk Storage

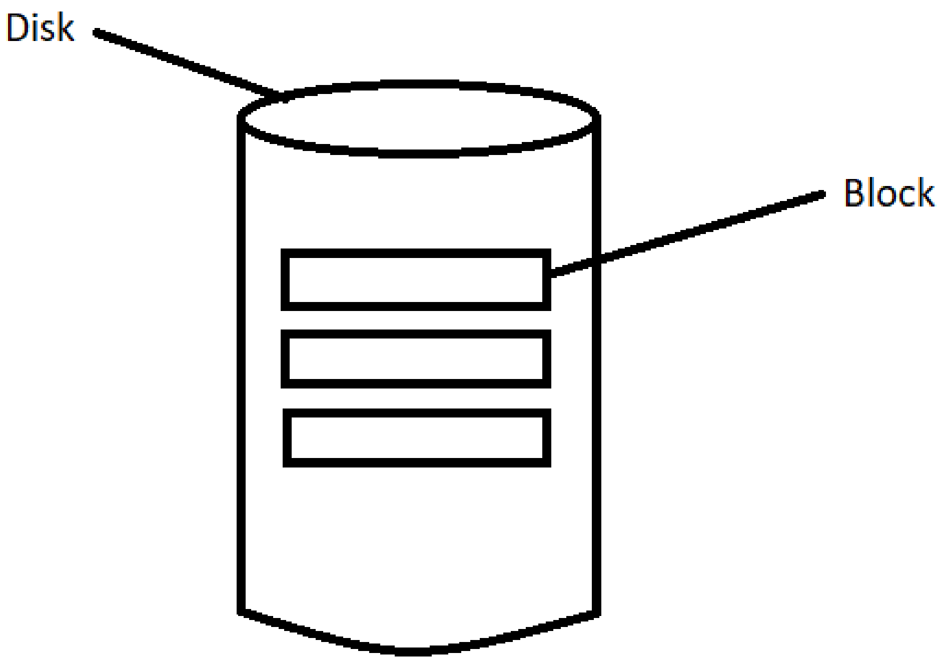
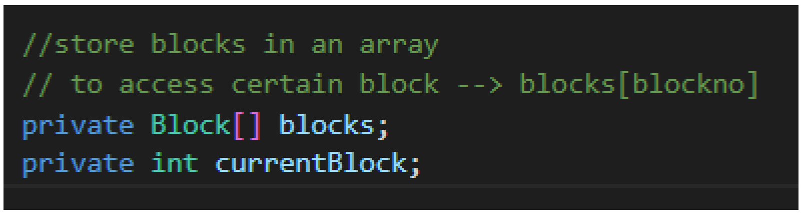


Figure 5: How blocks are stored in a disk

The blocks are then arranged sequentially on the disk to minimise seek and rotational delay.



```
//store blocks in an array
// to access certain block --> blocks[blockno]
private Block[] blocks;
private int currentBlock;
```

Figure 6: Block class

Similar to the diagram shown in Fig 5, Fig 6 shows how we implement the disk storage by using an array of Block objects to store the blocks and the index of the array represents the current block that we want to access. For e.g. when current block(int) = 4, blocks[4] allow us to access the 5th block in the disk.

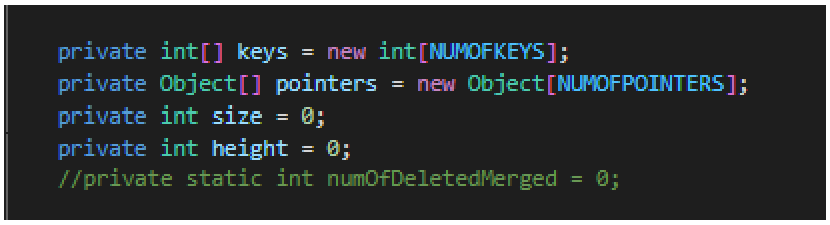
Disk Operations

The disk is able to support several operations:

- Insert Record
- Delete Record
- Search Record
- Print Record

Design of B+ Tree Component

Data Structure of a node



```
private int[] keys = new int[NUMOFKEYS];
private Object[] pointers = new Object[NUMOFPOINTERS];
private int size = 0;
private int height = 0;
//private static int numOfDeletedMerged = 0;
```

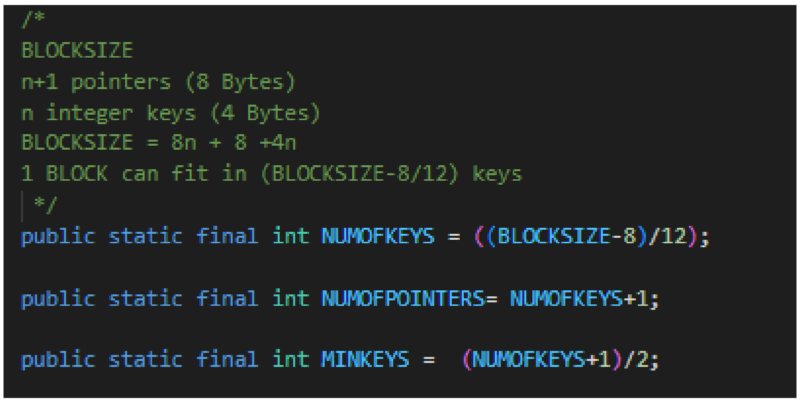
Figure 7: Variables of a node

Every Node consists of the following variables:

- Keys
    - o Integer Array of size (Bytes) that stores the (name) value
- Pointers
    - o Object Array of size (Bytes) that stores pointers address of each child/sibling nodes or (name) value

- Size
  - Size of the node
- Height
  - Height of the B+ Tree

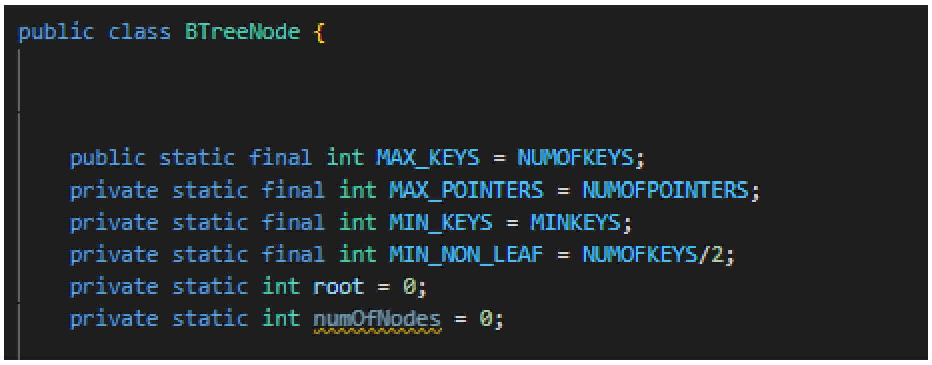
Maximum number of keys in which a Node maintains



```java
/*
BLOCKSIZE
n+1 pointers (8 Bytes)
n integer keys (4 Bytes)
BLOCKSIZE = 8n + 8 +4n
1 BLOCK can fit in (BLOCKSIZE-8/12) keys
 */
public static final int NUMOFKEYS = ((BLOCKSIZE-8)/12);

public static final int NUMOFPOINTERS= NUMOFKEYS+1;

public static final int MINKEYS =  (NUMOFKEYS+1)/2;
```

Figure 8: Functions in Utils.java

In Utils.java, functions are implemented to handle the calculation of values for respective variables. Fig 8 is how we calculate the number of keys, number of pointers and minimum number of keys in a node. As we know that the pointer size is 8bytes due to 64-bit processor and operating system, together with the key size which is 4 bytes for numVotes integer, we can know that every key-pointer pair will have a size of 12bytes in total. Hence, we are able to calculate the maximum number of keys that can be implemented in a node with regards to the block size. At the same time, we are also able to calculate the minimum number of keys in a leaf node with the formula which is the floor of (n+1)/2.



```java
public class BTreeNode {



    public static final int MAX_KEYS = NUMOFKEYS;
    private static final int MAX_POINTERS = NUMOFPOINTERS;
    private static final int MIN_KEYS = MINKEYS;
    private static final int MIN_NON_LEAF = NUMOFKEYS/2;
    private static int root = 0;
    private static int numOfNodes = 0;
```

Figure 9: Variables instantiated in BTreeNode class

With the values we get from Utils.java, the maximum number of keys (MAX_KEYS) can be fetched from it, which correlates to the size of the block.

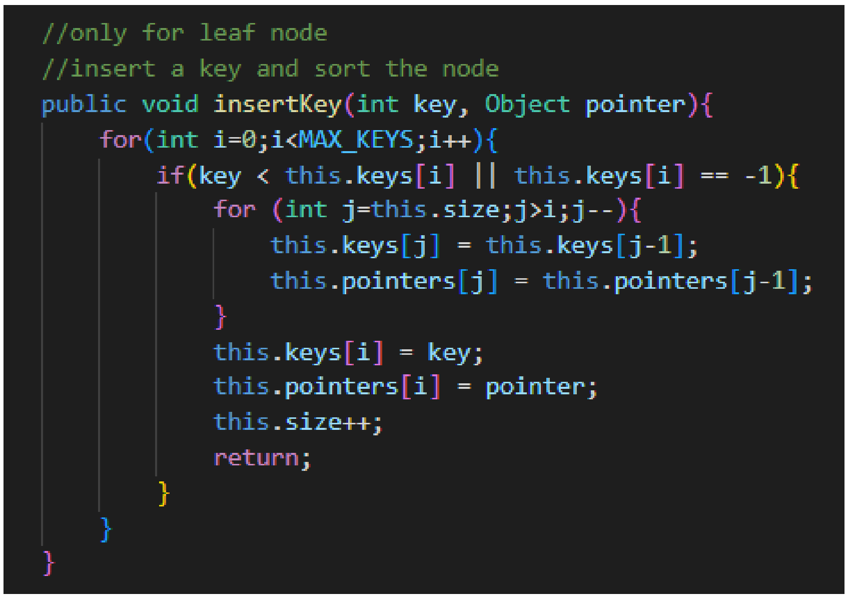
- For a block size of 200B, MAX_KEYS will be $\lfloor [(200 - 8) / (4 + 8)] \rfloor = 16$

- For a block size of 500B, MAX_KEYS will be of ⌊ [(500 – 8) / (4 + 8)] ⌋ = 41

Building of B+ Tree

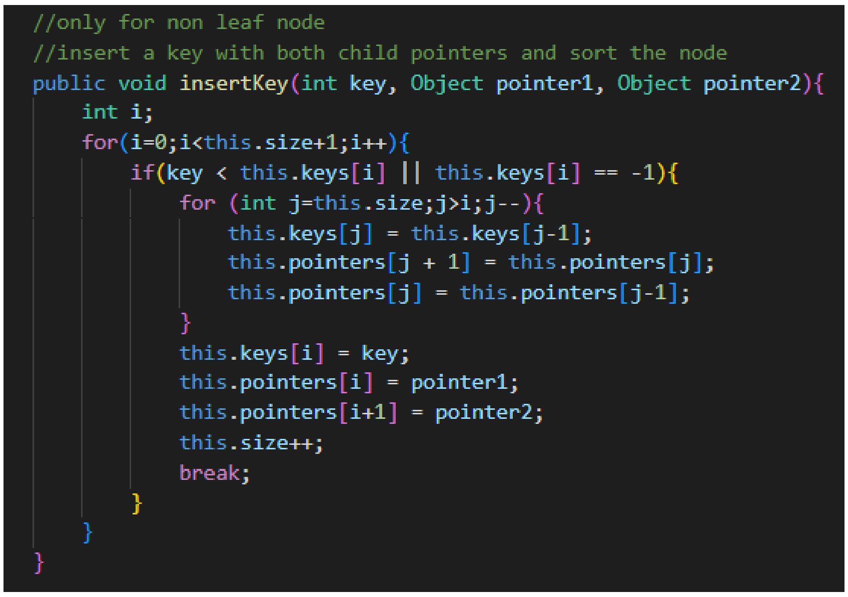
The implementation of B+ Tree is mostly done in BTreeNode.java. It supports the 3 main operations: Insert, Search and Delete.

<u>Insertion</u>



```java
//only for leaf node
//insert a key and sort the node
public void insertKey(int key, Object pointer){
    for(int i=0;i<MAX_KEYS;i++){
        if(key < this.keys[i] || this.keys[i] == -1){
            for (int j=this.size;j>i;j--){
                this.keys[j] = this.keys[j-1];
                this.pointers[j] = this.pointers[j-1];
            }
            this.keys[i] = key;
            this.pointers[i] = pointer;
            this.size++;
            return;
        }
    }
}
```

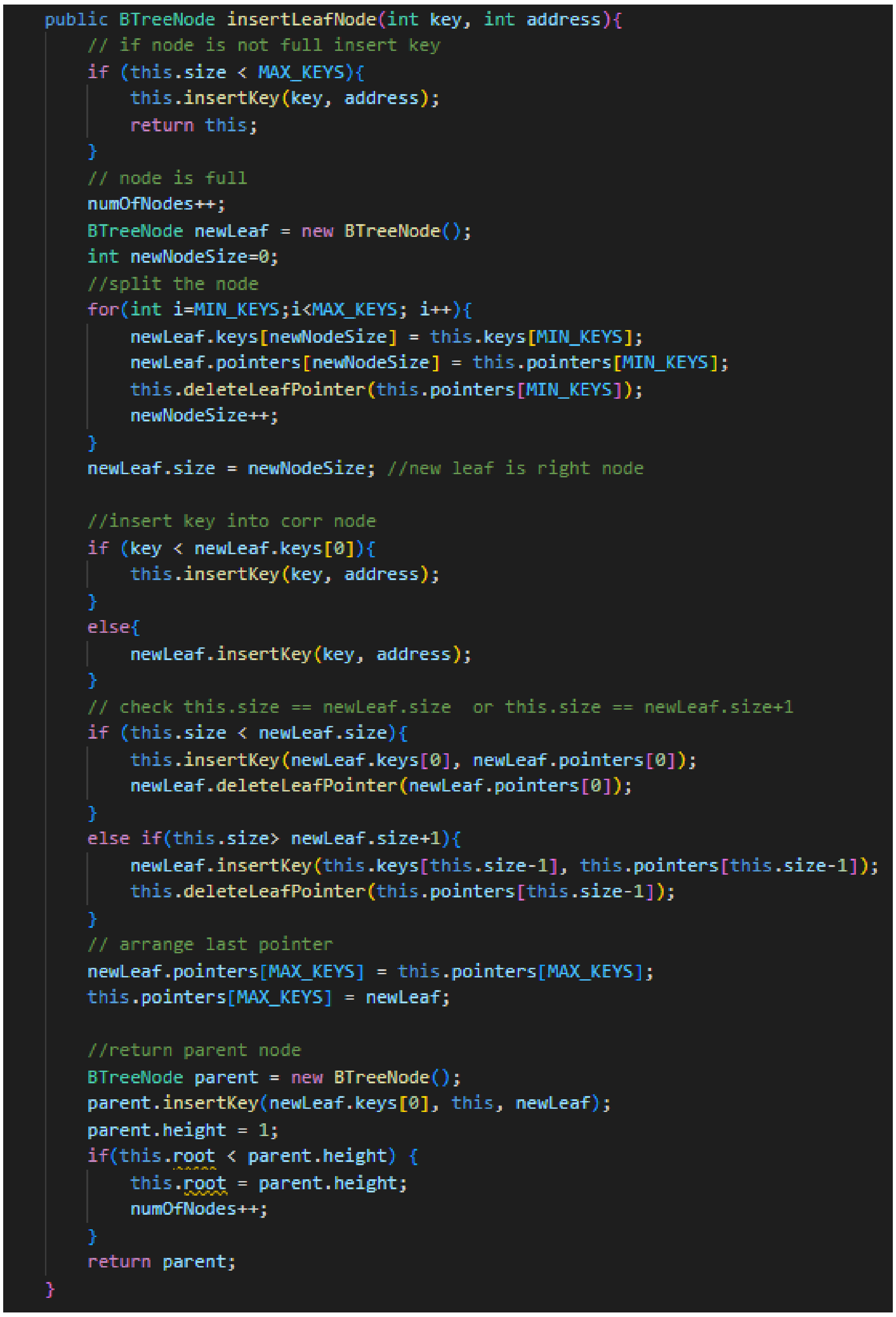
Figure 10: Base logic to handle insertion of keys in leaf node



```java
//only for non leaf node
//insert a key with both child pointers and sort the node
public void insertKey(int key, Object pointer1, Object pointer2){
    int i;
    for(i=0;i<this.size+1;i++){
        if(key < this.keys[i] || this.keys[i] == -1){
            for (int j=this.size;j>i;j--){
                this.keys[j] = this.keys[j-1];
                this.pointers[j + 1] = this.pointers[j];
                this.pointers[j] = this.pointers[j-1];
            }
            this.keys[i] = key;
            this.pointers[i] = pointer1;
            this.pointers[i+1] = pointer2;
            this.size++;
            break;
        }
    }
}
```

Figure 11:Base logic to handle insertion of keys in non-leaf node

Fig 10 and 11 shows the logic to handle insertion of key into both leaf and non-leaf nodes and to sort them after it is being inserted. By looping through the keys in the node, the new key inserted will be compared key by key. This is to check the actual position that it should be inserted within the node.

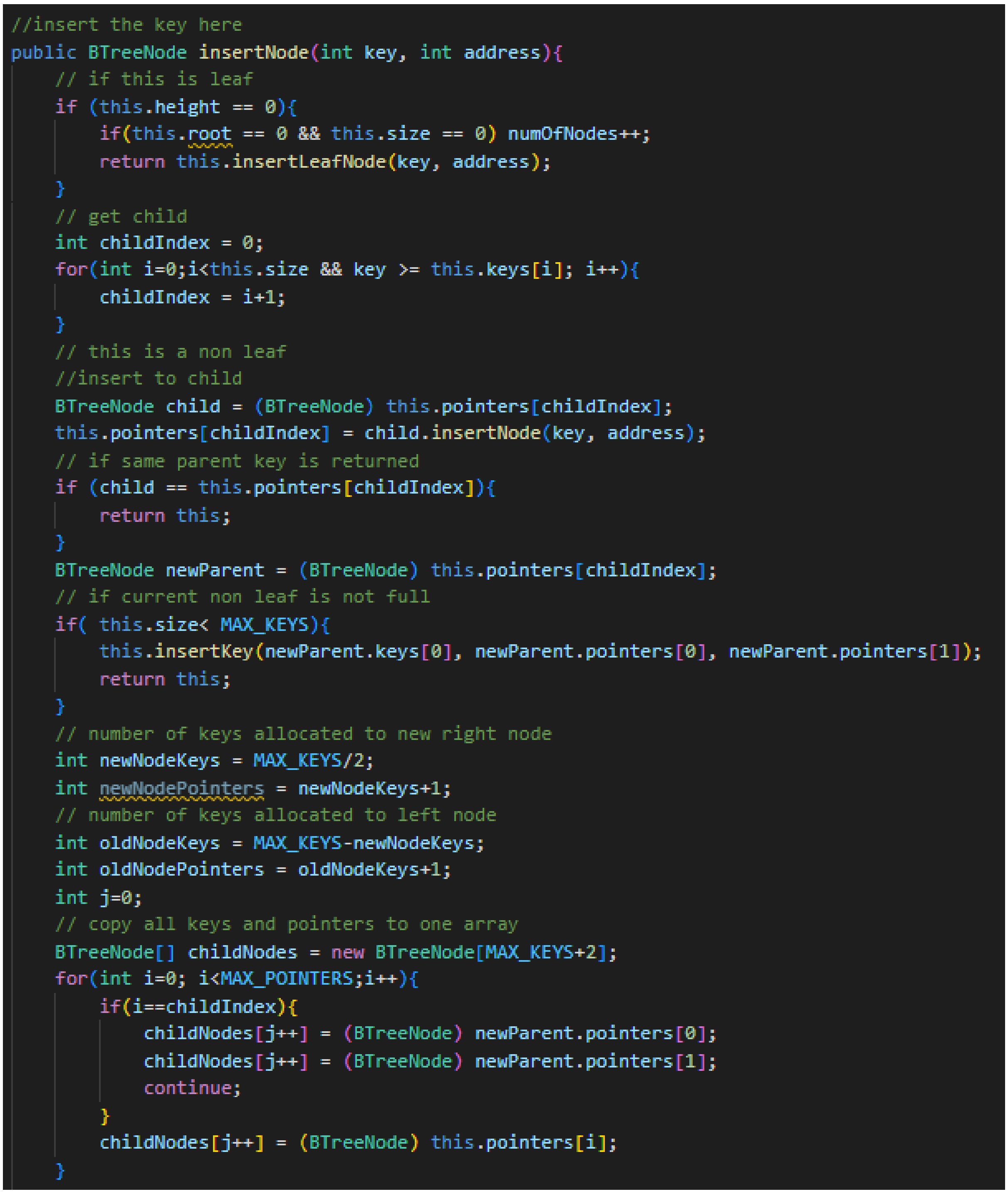


```java
public BTreeNode insertLeafNode(int key, int address){
    // if node is not full insert key
    if (this.size < MAX_KEYS){
        this.insertKey(key, address);
        return this;
    }
    // node is full
    numOfNodes++;
    BTreeNode newLeaf = new BTreeNode();
    int newNodeSize=0;
    //split the node
    for(int i=MIN_KEYS;i<MAX_KEYS; i++){
        newLeaf.keys[newNodeSize] = this.keys[MIN_KEYS];
        newLeaf.pointers[newNodeSize] = this.pointers[MIN_KEYS];
        this.deleteLeafPointer(this.pointers[MIN_KEYS]);
        newNodeSize++;
    }
    newLeaf.size = newNodeSize; //new leaf is right node

    //insert key into corr node
    if (key < newLeaf.keys[0]){
        this.insertKey(key, address);
    }
    else{
        newLeaf.insertKey(key, address);
    }
    // check this.size == newLeaf.size  or this.size == newLeaf.size+1
    if (this.size < newLeaf.size){
        this.insertKey(newLeaf.keys[0], newLeaf.pointers[0]);
        newLeaf.deleteLeafPointer(newLeaf.pointers[0]);
    }
    else if(this.size> newLeaf.size+1){
        newLeaf.insertKey(this.keys[this.size-1], this.pointers[this.size-1]);
        this.deleteLeafPointer(this.pointers[this.size-1]);
    }
    // arrange last pointer
    newLeaf.pointers[MAX_KEYS] = this.pointers[MAX_KEYS];
    this.pointers[MAX_KEYS] = newLeaf;

    //return parent node
    BTreeNode parent = new BTreeNode();
    parent.insertKey(newLeaf.keys[0], this, newLeaf);
    parent.height = 1;
    if(this.root < parent.height) {
        this.root = parent.height;
        numOfNodes++;
    }
    return parent;
}
```

Figure 12: Logic to handle insertion of keys into leaf node

Fig 12 shows how we implement the insertion of keys into a leaf node. If the leaf node still has empty space for insertion, we will just insert the key into the directly insert the key into the leaf node and return the node.

However, if we know that the insertion will cause an overflow, we will split the node and insert the second half of the node into a corresponding new node. We will then compare and check whether the key should be inserted in the left node or the right node. After inserting the key into the correct node, we will create a new parent node which holds the value of the minimum key of the right node and return it.

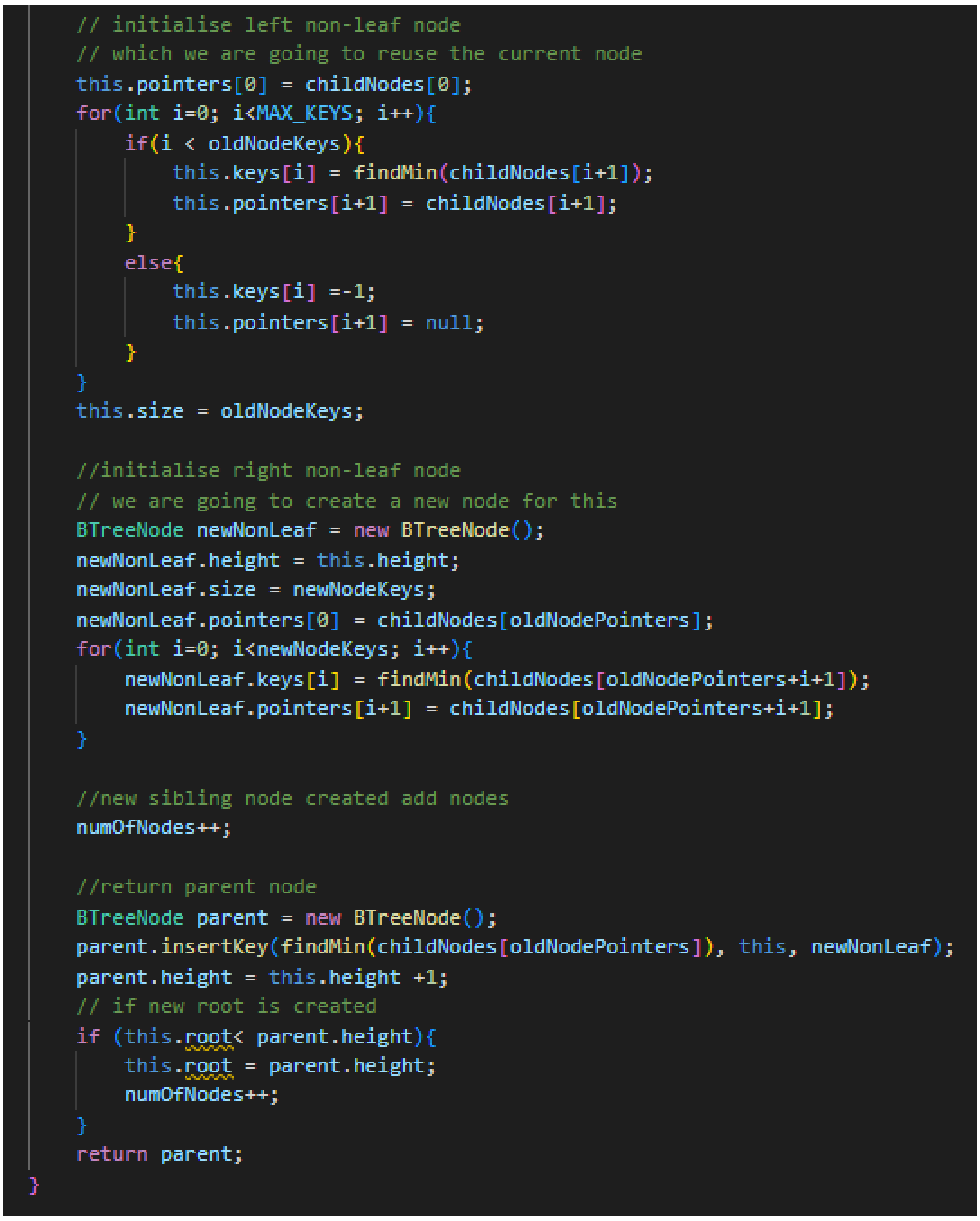


```java
//insert the key here
public BTreeNode insertNode(int key, int address){
    // if this is leaf
    if (this.height == 0){
        if(this.root == 0 && this.size == 0) numOfNodes++;
        return this.insertLeafNode(key, address);
    }
    // get child
    int childIndex = 0;
    for(int i=0;i<this.size && key >= this.keys[i]; i++){
        childIndex = i+1;
    }
    // this is a non leaf
    //insert to child
    BTreeNode child = (BTreeNode) this.pointers[childIndex];
    this.pointers[childIndex] = child.insertNode(key, address);
    // if same parent key is returned
    if (child == this.pointers[childIndex]){
        return this;
    }
    BTreeNode newParent = (BTreeNode) this.pointers[childIndex];
    // if current non leaf is not full
    if( this.size< MAX_KEYS){
        this.insertKey(newParent.keys[0], newParent.pointers[0], newParent.pointers[1]);
        return this;
    }
    // number of keys allocated to new right node
    int newNodeKeys = MAX_KEYS/2;
    int newNodePointers = newNodeKeys+1;
    // number of keys allocated to left node
    int oldNodeKeys = MAX_KEYS-newNodeKeys;
    int oldNodePointers = oldNodeKeys+1;
    int j=0;
    // copy all keys and pointers to one array
    BTreeNode[] childNodes = new BTreeNode[MAX_KEYS+2];
    for(int i=0; i<MAX_POINTERS;i++){
        if(i==childIndex){
            childNodes[j++] = (BTreeNode) newParent.pointers[0];
            childNodes[j++] = (BTreeNode) newParent.pointers[1];
            continue;
        }
        childNodes[j++] = (BTreeNode) this.pointers[i];
    }
```

Figure 13(i): Logic to handle insertion of keys into the B+ Tree (Part 1)

Fig 13(i) shows the first part of how we implement the insertion of keys into the B+ Tree. If we know that the tree is on the first level by checking the height, we know that it is a leaf node and call the insertion of keys into the leaf node as shown in Fig 12.

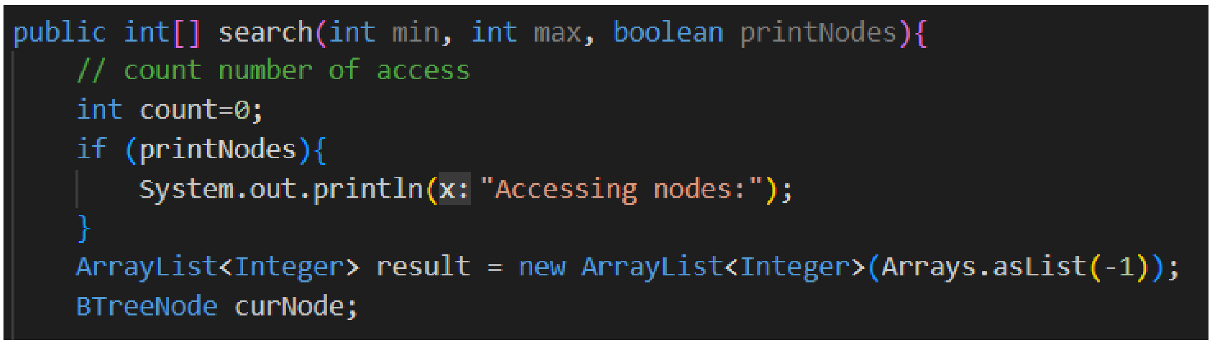
If it is not the first level, the function will retrieve the child nodes first and try to insert the keys into the child node with the same parents. The number of keys will be updated and allocated to the left node and the new right node. All the keys and pointers will then be copied into an array.



```
        // initialise left non-leaf node
        // which we are going to reuse the current node
        this.pointers[0] = childNodes[0];
        for(int i=0; i<MAX_KEYS; i++){
            if(i < oldNodeKeys){
                this.keys[i] = findMin(childNodes[i+1]);
                this.pointers[i+1] = childNodes[i+1];
            }
            else{
                this.keys[i] =-1;
                this.pointers[i+1] = null;
            }
        }
        this.size = oldNodeKeys;

        //initialise right non-leaf node
        // we are going to create a new node for this
        BTreeNode newNonLeaf = new BTreeNode();
        newNonLeaf.height = this.height;
        newNonLeaf.size = newNodeKeys;
        newNonLeaf.pointers[0] = childNodes[oldNodePointers];
        for(int i=0; i<newNodeKeys; i++){
            newNonLeaf.keys[i] = findMin(childNodes[oldNodePointers+i+1]);
            newNonLeaf.pointers[i+1] = childNodes[oldNodePointers+i+1];
        }

        //new sibling node created add nodes
        numOfNodes++;

        //return parent node
        BTreeNode parent = new BTreeNode();
        parent.insertKey(findMin(childNodes[oldNodePointers]), this, newNonLeaf);
        parent.height = this.height +1;
        // if new root is created
        if (this.root< parent.height){
            this.root = parent.height;
            numOfNodes++;
        }
        return parent;
    }
```

Figure 13(ii): Logic to handle insertion of keys into B+ Tree (Part 2)

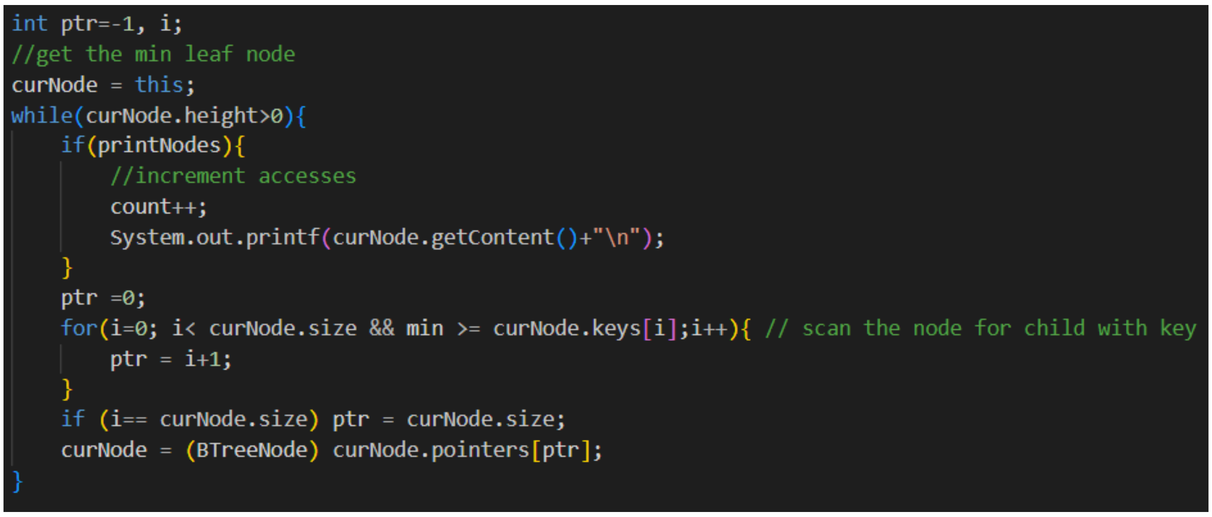
Searching



```java
public int[] search(int min, int max, boolean printNodes){
    // count number of access
    int count=0;
    if (printNodes){
        System.out.println(x: "Accessing nodes:");
    }
    ArrayList<Integer> result = new ArrayList<Integer>(Arrays.asList(-1));
    BTreeNode curNode;
```

Figure 14: Searching of leaf nodes (Declaration)

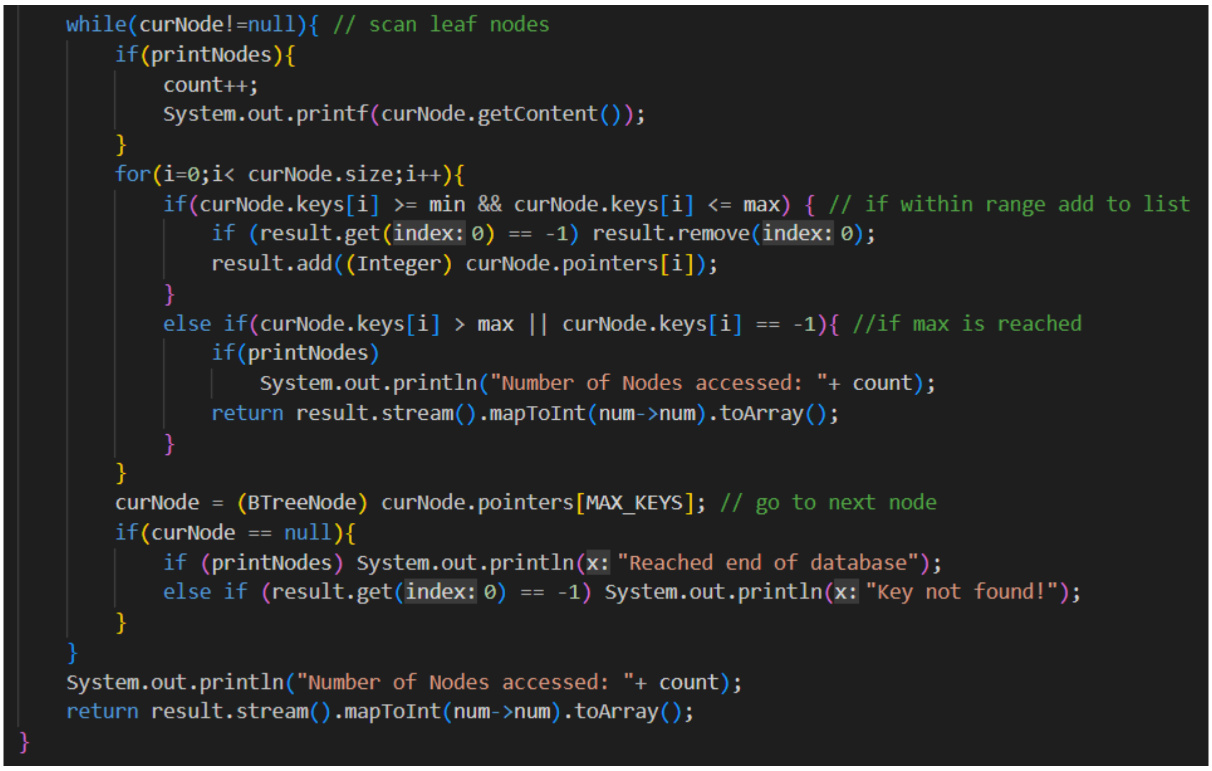
Fig 14 shows the logic to handle searching of both leaf and non-leaf nodes. Firstly, an ArrayList is declared in order to store the results and the BTreeNode is called instantiated in order to traverse down the B+ tree.



```java
int ptr=-1, i;
//get the min leaf node
curNode = this;
while(curNode.height>0){
    if(printNodes){
        //increment accesses
        count++;
        System.out.printf(curNode.getContent()+"\n");
    }
    ptr =0;
    for(i=0; i< curNode.size && min >= curNode.keys[i];i++){ // scan the node for child with key
        ptr = i+1;
    }
    if (i== curNode.size) ptr = curNode.size;
    curNode = (BTreeNode) curNode.pointers[ptr];
}
```

Figure 15: Searching of leaf nodes (Finding the leaf Node)

Fig 15 shows the continuation of the logic to handle searching of both leaf and non-leaf nodes. To traverse down the B+ tree, a loop is used against the height of the tree to recursively access the minimum leaf node. To ensure that the minimum leaf node is accessed, the count of the traversing is checked against the size of the current node.

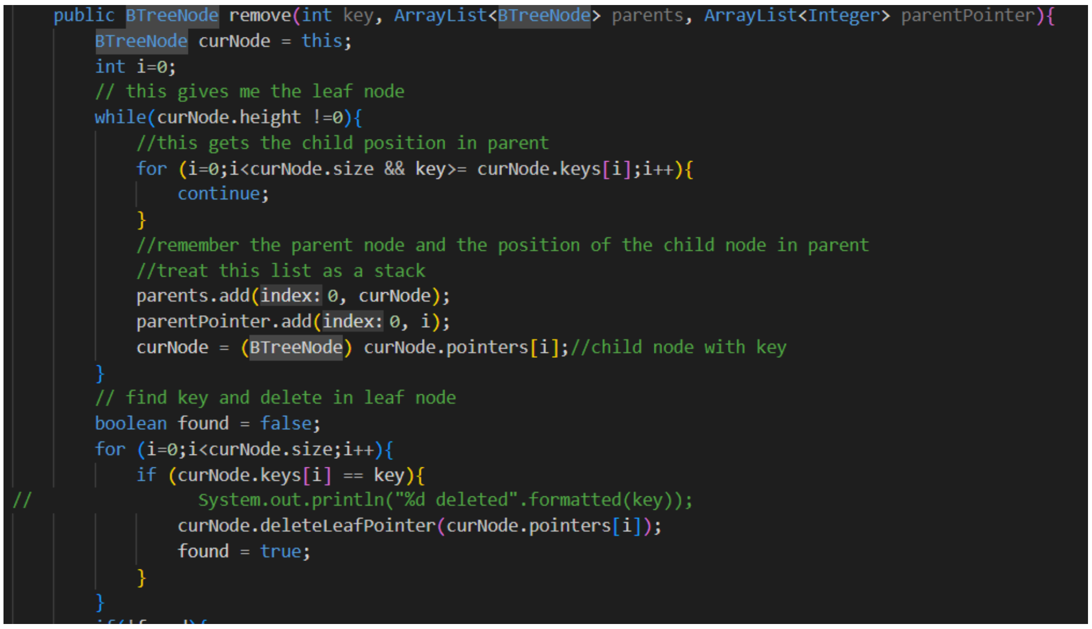


```
    while(curNode!=null){ // scan leaf nodes
        if(printNodes){
            count++;
            System.out.printf(curNode.getContent());
        }
        for(i=0;i< curNode.size;i++){
            if(curNode.keys[i] >= min && curNode.keys[i] <= max) { // if within range add to list
                if (result.get(index: 0) == -1) result.remove(index: 0);
                result.add((Integer) curNode.pointers[i]);
            }
            else if(curNode.keys[i] > max || curNode.keys[i] == -1){ //if max is reached
                if(printNodes)
                    System.out.println("Number of Nodes accessed: "+ count);
                return result.stream().mapToInt(num->num).toArray();
            }
        }
        curNode = (BTreeNode) curNode.pointers[MAX_KEYS]; // go to next node
        if(curNode == null){
            if (printNodes) System.out.println(x: "Reached end of database");
            else if (result.get(index: 0) == -1) System.out.println(x: "Key not found!");
        }
    }
    System.out.println("Number of Nodes accessed: "+ count);
    return result.stream().mapToInt(num->num).toArray();
}
```

Figure 16: Searching of leaf nodes (Finding the leaf Node)

Fig 16 shows the continuation of the logic to handle searching of both leaf and non-leaf nodes. While accessing the leaf nodes of the B+ tree, the key value of the leaf nodes are checked to see if it is within the range. If the values are indeed within the range of the leaf nodes, the key-value pair will be pushed into the ArrayList<result> and return as the output. If the values exceeds the maximum value of the leaf node, it will be redirected to access the subsequent leaf node until the value could be found within the list range. After all the leaf nodes have been accessed and if the values could not be found, search() will return a message to indicate the end of the database with no key found.

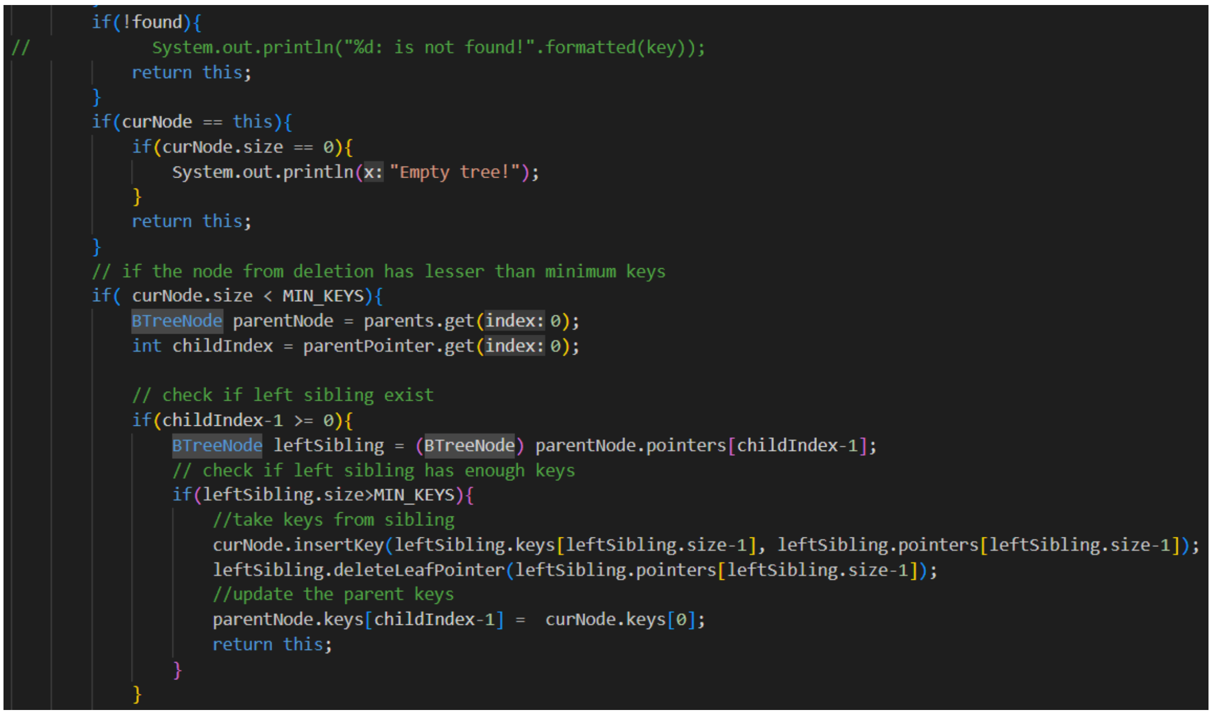
Deletion



```
    public BTreeNode remove(int key, ArrayList<BTreeNode> parents, ArrayList<Integer> parentPointer){
        BTreeNode curNode = this;
        int i=0;
        // this gives me the leaf node
        while(curNode.height !=0){
            //this gets the child position in parent
            for (i=0;i<curNode.size && key>= curNode.keys[i];i++){
                continue;
            }
            //remember the parent node and the position of the child node in parent
            //treat this list as a stack
            parents.add(index: 0, curNode);
            parentPointer.add(index: 0, i);
            curNode = (BTreeNode) curNode.pointers[i];//child node with key
        }
        // find key and delete in leaf node
        boolean found = false;
        for (i=0;i<curNode.size;i++){
            if (curNode.keys[i] == key){
//                  System.out.println("%d deleted".formatted(key));
                curNode.deleteLeafPointer(curNode.pointers[i]);
                found = true;
            }
        }
```

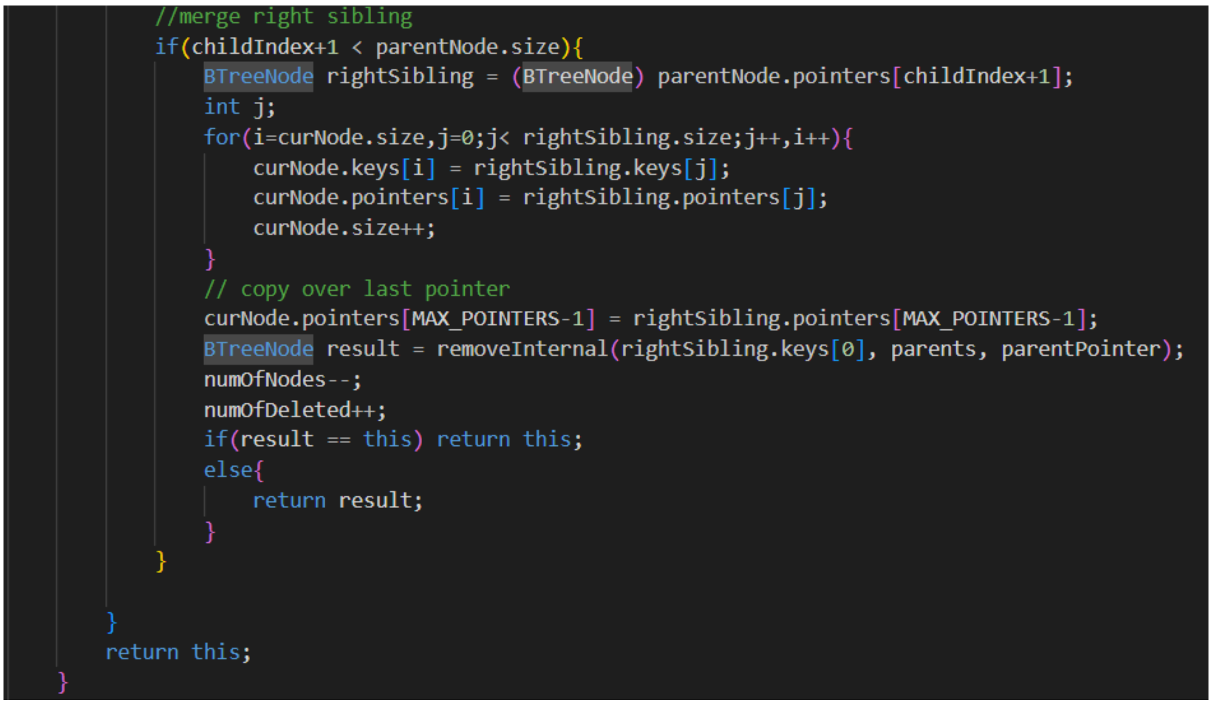
Figure 17: Remove of leaf nodes (Instantiation & Removal)

Fig 17 shows the logic of the removal of leaf nodes. It requires the parameters of the specified key, the arraylist in which the key belongs to, and its parent pointer. The function as stated stores the parent node and the position of the child node with relevance to the parent node. If the key is found, it will then be successfully removed.



```
        if(!found){
//              System.out.println("%d: is not found!".formatted(key));
            return this;
        }
        if(curNode == this){
            if(curNode.size == 0){
                System.out.println(x: "Empty tree!");
            }
            return this;
        }
        // if the node from deletion has lesser than minimum keys
        if( curNode.size < MIN_KEYS){
            BTreeNode parentNode = parents.get(index: 0);
            int childIndex = parentPointer.get(index: 0);

            // check if left sibling exist
            if(childIndex-1 >= 0){
                BTreeNode leftSibling = (BTreeNode) parentNode.pointers[childIndex-1];
                // check if left sibling has enough keys
                if(leftSibling.size>MIN_KEYS){
                    //take keys from sibling
                    curNode.insertKey(leftSibling.keys[leftSibling.size-1], leftSibling.pointers[leftSibling.size-1]);
                    leftSibling.deleteLeafPointer(leftSibling.pointers[leftSibling.size-1]);
                    //update the parent keys
                    parentNode.keys[childIndex-1] =  curNode.keys[0];
                    return this;
                }
            }
```

Figure 18.1: Remove of leaf nodes (Handling Errors)



```
            //merge right sibling
            if(childIndex+1 < parentNode.size){
                BTreeNode rightSibling = (BTreeNode) parentNode.pointers[childIndex+1];
                int j;
                for(i=curNode.size,j=0;j< rightSibling.size;j++,i++){
                    curNode.keys[i] = rightSibling.keys[j];
                    curNode.pointers[i] = rightSibling.pointers[j];
                    curNode.size++;
                }
                // copy over last pointer
                curNode.pointers[MAX_POINTERS-1] = rightSibling.pointers[MAX_POINTERS-1];
                BTreeNode result = removeInternal(rightSibling.keys[0], parents, parentPointer);
                numOfNodes--;
                numOfDeleted++;
                if(result == this) return this;
                else{
                    return result;
                }
            }

        }
        return this;
    }
```

Figure 18.2: Remove of leaf nodes (Deletion & Combination of Left Sibling)

Fig 18.1 and 18.2 shows the continuation of the removal logic of leaf nodes. If the key is not found in the array list, an error message will be returned back to the user. An error message will also return back to the user if the user decides to remove a value from the empty tree. Once a node is removed, it has to be ensured that the size of the leaf nodes will be above the minimum threshold of the minimum key. If the number of nodes fall below the threshold, it will check if any left sibling nodes exist, move the keys from the sibling nodes and update the parent keys. The nodes will then be merged to the right sibling and the current node will copy over the last pointer.

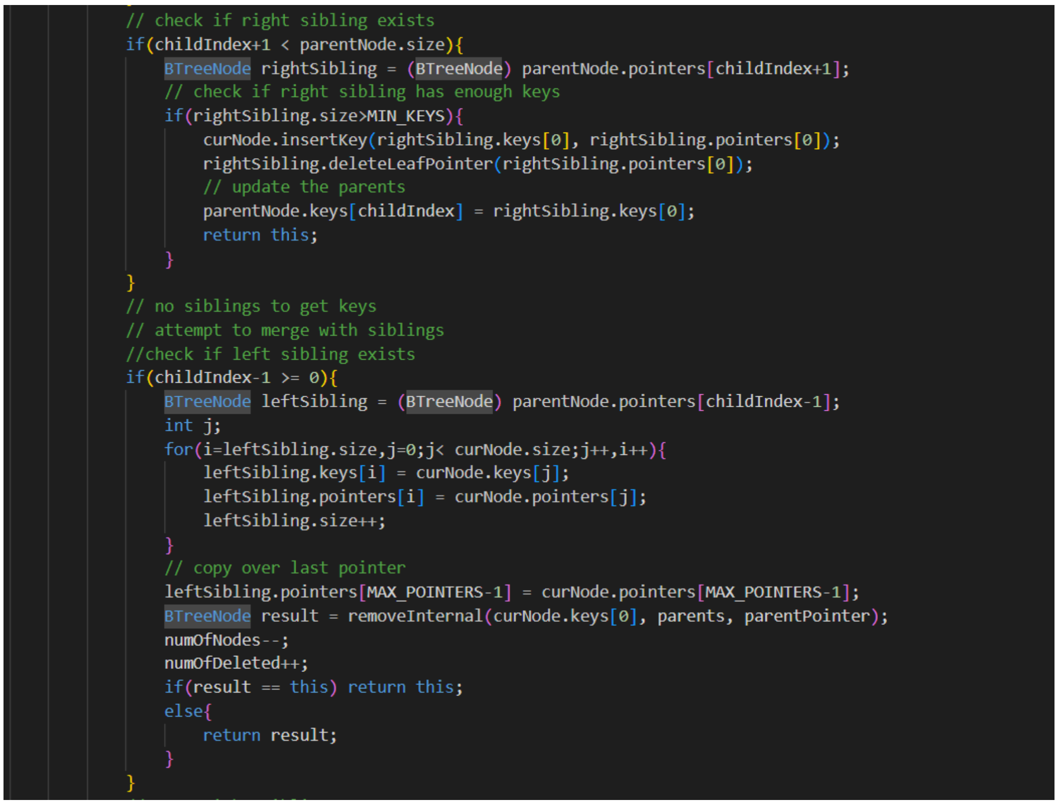

```
// check if right sibling exists
if(childIndex+1 < parentNode.size){
    BTreeNode rightSibling = (BTreeNode) parentNode.pointers[childIndex+1];
    // check if right sibling has enough keys
    if(rightSibling.size>MIN_KEYS){
        curNode.insertKey(rightSibling.keys[0], rightSibling.pointers[0]);
        rightSibling.deleteLeafPointer(rightSibling.pointers[0]);
        // update the parents
        parentNode.keys[childIndex] = rightSibling.keys[0];
        return this;
    }
}
// no siblings to get keys
// attempt to merge with siblings
//check if left sibling exists
if(childIndex-1 >= 0){
    BTreeNode leftSibling = (BTreeNode) parentNode.pointers[childIndex-1];
    int j;
    for(i=leftSibling.size,j=0;j< curNode.size;j++,i++){
        leftSibling.keys[i] = curNode.keys[j];
        leftSibling.pointers[i] = curNode.pointers[j];
        leftSibling.size++;
    }
    // copy over last pointer
    leftSibling.pointers[MAX_POINTERS-1] = curNode.pointers[MAX_POINTERS-1];
    BTreeNode result = removeInternal(curNode.keys[0], parents, parentPointer);
    numOfNodes--;
    numOfDeleted++;
    if(result == this) return this;
    else{
        return result;
    }
}
```

Figure 19: Remove of leaf nodes (Combination of Right Sibling)

Fig 19 shows the continuation of the removal logic of leaf nodes. Once the left sibling node has been checked and verified, the steps taken are repeated for the right sibling node.

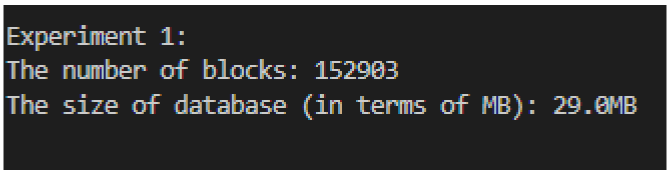
Experiment Results

Experiment 1 Results

For Experiment 1, our team has stored the data on the disk and fetched the respective outputs:

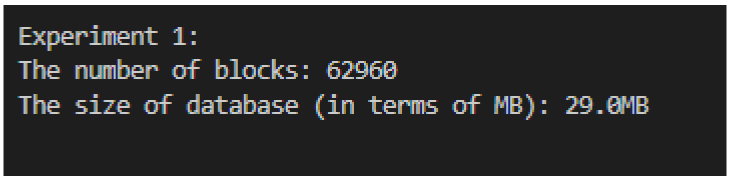
1. Number of Blocks
2. Size of the Database (in terms of MB)

Block Size	Number Of Blocks	Database Size
200 B	152903	29.0 MB
500 B	62960	29.0 MB



```
Experiment 1:
The number of blocks: 152903
The size of database (in terms of MB): 29.0MB
```

Figure 20: Output for Experiment 1 for 200MB



```
Experiment 1:
The number of blocks: 62960
The size of database (in terms of MB): 29.0MB
```

Figure 21: Output for Experiment 1 for 500MB

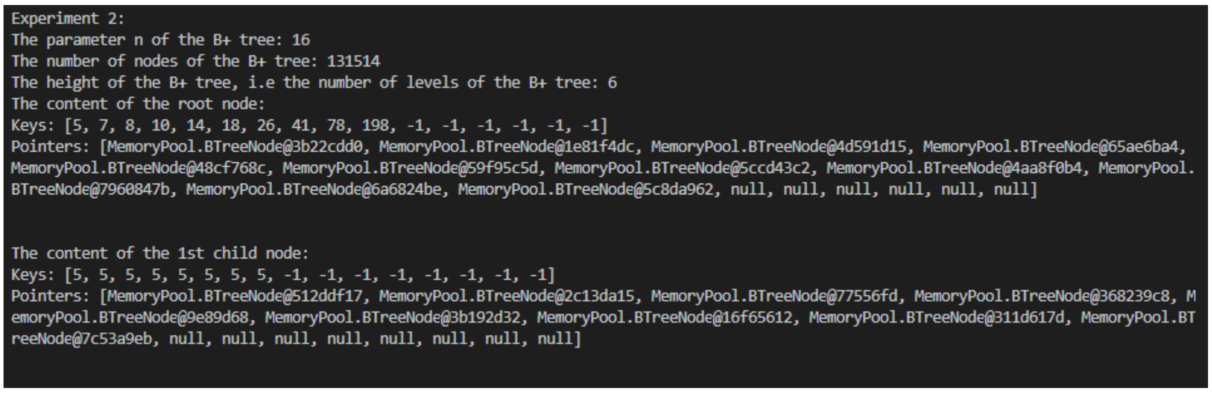
Experiment 2 Results

For Experiment 2, our team has built the B+ tree based on the attribute “numVotes” by inserting the records sequentially and reported on the following statistics:

B+ Tree

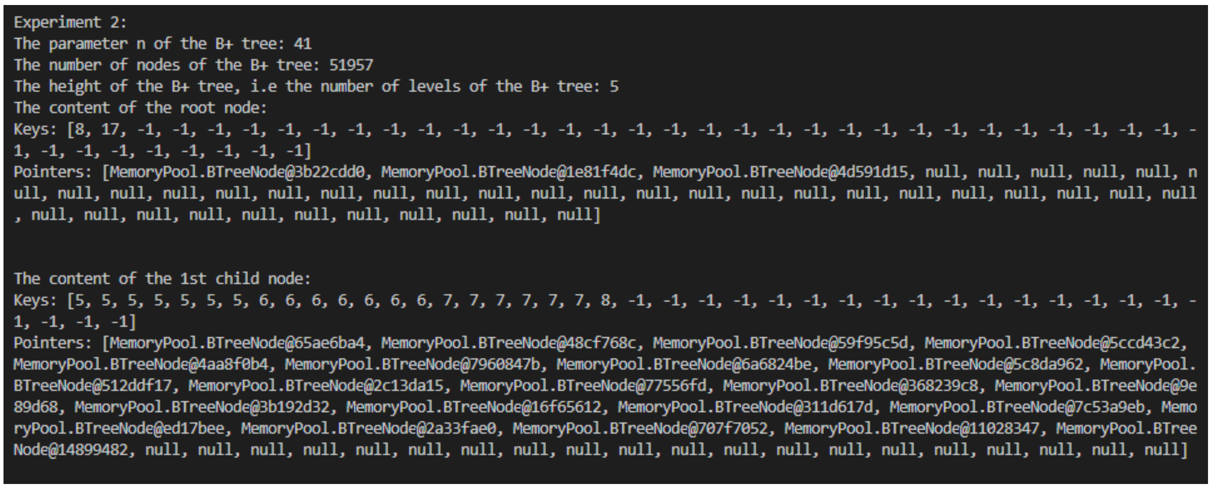
1. Parameter ***n*** of B+ tree
2. Number of Nodes of B+ tree
3. Height of the B+ tree
4. Content of the root node and its 1st child node

B+ Tree				
Block Size	**n**	**Number Of Nodes**	**Height**	**Content for root node & 1st child node**
200 B	16	131514	6	Refer to Figure 22
500 B	41	51957	5	Refer to Figure 23



```
Experiment 2:
The parameter n of the B+ tree: 16
The number of nodes of the B+ tree: 131514
The height of the B+ tree, i.e the number of levels of the B+ tree: 6
The content of the root node:
Keys: [5, 7, 8, 10, 14, 18, 26, 41, 78, 198, -1, -1, -1, -1, -1, -1]
Pointers: [MemoryPool.BTreeNode@3b22cdd0, MemoryPool.BTreeNode@1e81f4dc, MemoryPool.BTreeNode@4d591d15, MemoryPool.BTreeNode@65ae6ba4,
MemoryPool.BTreeNode@48cf768c, MemoryPool.BTreeNode@59f95c5d, MemoryPool.BTreeNode@5ccd43c2, MemoryPool.BTreeNode@4aa8f0b4, MemoryPool.
BTreeNode@7960847b, MemoryPool.BTreeNode@6a6824be, MemoryPool.BTreeNode@5c8da962, null, null, null, null, null, null]


The content of the 1st child node:
Keys: [5, 5, 5, 5, 5, 5, 5, 5, -1, -1, -1, -1, -1, -1, -1, -1]
Pointers: [MemoryPool.BTreeNode@512ddf17, MemoryPool.BTreeNode@2c13da15, MemoryPool.BTreeNode@77556fd, MemoryPool.BTreeNode@368239c8, M
emoryPool.BTreeNode@9e89d68, MemoryPool.BTreeNode@3b192d32, MemoryPool.BTreeNode@16f65612, MemoryPool.BTreeNode@311d617d, MemoryPool.BT
reeNode@7c53a9eb, null, null, null, null, null, null, null, null]
```

Figure 22: Experiment 2 output for block size 200MB



```
Experiment 2:
The parameter n of the B+ tree: 41
The number of nodes of the B+ tree: 51957
The height of the B+ tree, i.e the number of levels of the B+ tree: 5
The content of the root node:
Keys: [8, 17, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -
1, -1, -1, -1, -1, -1, -1, -1, -1]
Pointers: [MemoryPool.BTreeNode@3b22cdd0, MemoryPool.BTreeNode@1e81f4dc, MemoryPool.BTreeNode@4d591d15, null, null, null, null, null, n
ull, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null
, null, null, null, null, null, null, null, null, null, null, null]


The content of the 1st child node:
Keys: [5, 5, 5, 5, 5, 5, 5, 6, 6, 6, 6, 6, 6, 6, 7, 7, 7, 7, 7, 7, 8, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -
1, -1, -1, -1]
Pointers: [MemoryPool.BTreeNode@65ae6ba4, MemoryPool.BTreeNode@48cf768c, MemoryPool.BTreeNode@59f95c5d, MemoryPool.BTreeNode@5ccd43c2,
MemoryPool.BTreeNode@4aa8f0b4, MemoryPool.BTreeNode@7960847b, MemoryPool.BTreeNode@6a6824be, MemoryPool.BTreeNode@5c8da962, MemoryPool.
BTreeNode@512ddf17, MemoryPool.BTreeNode@2c13da15, MemoryPool.BTreeNode@77556fd, MemoryPool.BTreeNode@368239c8, MemoryPool.BTreeNode@9e
89d68, MemoryPool.BTreeNode@3b192d32, MemoryPool.BTreeNode@16f65612, MemoryPool.BTreeNode@311d617d, MemoryPool.BTreeNode@7c53a9eb, Memo
ryPool.BTreeNode@ed17bee, MemoryPool.BTreeNode@2a33fae0, MemoryPool.BTreeNode@707f7052, MemoryPool.BTreeNode@11028347, MemoryPool.BTree
Node@14899482, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null, null]
```

Figure 23: Experiment 2 output for blocksize 500MB

Experiment 3 Results

For experiment 3, we have to retrieve those movies with the “numVotes” equal to 500 and report the following statistics:

1. Number and Content of index nodes the process accesses
2. Number and Content of data blocks the process accesses
3. Average of “averageRating’s” of the records that are returned

		Block Size	
		200 B	500 B
Index Nodes Accessed	**Number**	19	10
	Content of index nodes and data blocks	Refer to the outputs in the Terminal	Refer to the outputs in the Terminal
Data Blocks Accessed	**Number**	110	110
	Content of index nodes and data blocks	Refer to outputs in the Terminal	Refer to the outputs in the Terminal
Average of “averageRating’s” of Records Returned		6.731818214329806	6.731818214329806

Experiment 4 Results

For experiment 4, our team has retrieved the movies with the attribute “numVotes” from 30,000 to 40,000, both inclusively and reported the following statistics:

1. Number and Content of index nodes the process accesses
2. Number and Content of data blocks the process accesses
3. Average of “averageRating’s” of the records that are returned

		Block Size	
		200 B	500 B
Index Nodes Accessed	**Number**	91	38
	Content	Refer to outputs in Terminal	Refer to outputs in Terminal
Data Blocks Accessed	**Number**	942	930
	Content	Refer to outputs in Terminal	Refer to outputs in Terminal
Average of “averageRating’s” of Records Returned		6.727911862221244	6.727911862221244

Experiment 5 Results

For experiment 5, our team has deleted the movies with the attribute “numVotes” equal to 1,000, updated the B+ tree accordingly, and reported the following statistics:

1. Number of times that a node is deleted during the process of updating the B+ tree
2. Number nodes of the updated B+ tree
3. Height of the updated B+ tree
4. Content of the root node and its 1st child node of the updated B+ tree

B+ Tree					
				Content	
Block Size	**Number Of Times A Node Is Deleted**	**Number Of Nodes**	**Height**	**Root Node**	**First Child of Root Node**
200 B	3	131511	5	Refer to outputs in Terminal	
500 B	6	263016	5	Refer to outputs in Terminal	