Faculty of Computers and Artificial Intelligence

Cairo University

Final Assessment Project

Course Title: Data storage and retrieval
Course Code: IS313
Semester: Second Semester
Date: May 2020
Instructor: Dr. Ayman Ramadan El-Kelany

Project 1: B-Tree Index on Hard-Disk

Prepared by:

Menna Allah Tarek Farouk	20170297
Mai Mostafa Abdelrahman	20170303
Mohamed Ahmed Abdelnabey	20170364
Abdelrahman Mohamed Ahmed	20170148
Kareem Ahmed Eltemsah	20170195

Chapter 1. Introduction

A) Description

Here in this project, we implemented the B-tree index using Java. The project mainly focuses on the operations of insertion, deletion, searching of a node. The structure of the project is based on files on Hard disk. So here we work on an index file named as `"B-TreeIndex.bin"`, which is the index file. The file can contain N pages, where each page can contain up to m descendants. Each descendant is a node that has a key and reference.

Each page begins with an integer called 'leafIndicator'. If its value is 0 this means the page is a leaf, otherwise it's a parent page with pointers to some other pages. The rest of the page is m*2 integers, where each pair describes the key and the reference. The project has mainly 5 required functions that cover the whole functionality of the project. These five functions are:

```
1. void CreateIndexFileFile (String filename, int numberOfRecords, int m)
2. int InsertNewRecordAtIndex (String filename, int RecordID, int Reference)
3. boolean DeleteRecordFromIndex (String filename, int RecordID)
4. void DisplayIndexFileContent (String filename)
5. int SearchARecord (String filename, int RecordID)
```

Now let's describe each one of them separately.

1) `void CreateIndexFileFile (String filename, int numberOfRecords, int m)`

This function creates a file with numberOfRecords pages, and each page contains 2*m +1 integers. The first page is a guide page and no insertions are done there. The second integer of it represents the index of first empty page in the file. As long as the page is empty, it contains reference to the next empty page in the file, except the last page, where it should always contain -1. So initially, using a loop we count from 1 to (numberOfRecords - 2) and write these numbers as reference to empty pages. Then we write the keys and offsets as -1.

2) `int InsertNewRecordAtIndex (String filename, int RecordID, int Reference)`

This function inserts a new node with RecordID and Reference in file named as filename and returns the index of page where it's finally inserted. There're many situations for inserting a node in a tree. These situations are listed below:

a. Insert a new node at the empty tree.
b. Insert a new node at a leaf page and this page is not complete.
c. Insert a new node at a leaf page and this page is complete and its parent is not complete.
d. Insert a new node at a leaf page and this page is complete and its parent is also complete.
e. Insert a new node at the root page, while it's leaf and complete.

Now let's express each situation separately.

a. Insert a new node at the empty tree

This situation follows the following algorithms:

1- seek the file pointer to (4*(2*m + 1))
2- update the leaf indicator by writing 0
3- read the next empty place and assign its value to **nextEmptyPlace**
4- seek the file pointer back to (4 + 4*(2*m + 1))
5- write the new key and reference
6- seek the file pointer to (4)
7- update the first empty place with **nextEmptyPlace**
8- return 1

b. Insert a new node at a leaf page and this page is not complete

This situation is delegated to this function:

```
int insertAtNonCompletePage (RandomAccessFile file, Page page, Node node, int pageIndex)
```

which follows the following algorithms:

1- if page. **leafIndicator** == -1
  1.1- read the value of next empty page and assign it to **next**
  1.2- seek file pointer to (4) and write **next**
2- insert the node at the correct index in the page
3- seek the file pointer to (4 * **pageIndex** * (2 * m + 1))
4- write the page at the current offset
5- if the new key > the largest key before insertion
  5.1- update all instances of the largest key in the parent pages with the new key
6- return **pageIndex**

c. Insert a new node at a leaf page and this page is complete and its parent is not complete

This situation is delegated to this function:

```
insertAndSplitCurrentAndMakeNewParent (RandomAccessFile file, Node node,
Page oldPage, int pageIndex)
```

which follows the following algorithms:

1- insert the node at the correct index
2- set **middle** = the node at middle of the page nodes, **last** = the largest node
3- add all nodes <= **middle** to **leftRecords**
4- add all nodes > **middle** to **rightRecords**
5- seek to (4) and set **nextEmptyPlace** = first empty place
6- seek to (4 * next * (2* m +1)) , read integer and assign it to **newEmptyPlace**
7- clear all nodes of the page, add **leftRecords** to it
8- seek to (**pageIndex** * 4 * (2 * m + 1)) , and write the page at this offset
9- make **newPage**, add the **rightRecords**
10- seek to (**nextEmptyPlace** * 4 *(2*m + 1)) , and write **newPage** there
11- go to the parent page, remove the node that represents the old page
12- add **middle** with reference to the current page
13- add **last** with reference to **newPage**
14- seek to (4), and write **newEmptyPlace** there as the first empty place

d. Insert a new node at a leaf page and this page is complete and its parent is also complete.

This situation is delegated to this function:

```
int insertAndMakeNewLevelOfParents (RandomAccessFile file, Node node, Page page, int pageIndex, int parentPageIndex)
```
which follows the following algorithms:

1- insert the node at the correct index
2- set **middle** = the node at middle of the page nodes, **last** = the largest node
3- add all nodes <= **middle** to **leftRecords**
4- add all nodes > **middle** to **rightRecords**
5- seek to (4) and set **firstEmptyPlace** = first empty place
6- seek to (4 * **firstEmptyPlace** * (2* m +1)) , read integer and assign it to **secondEmptyPlace**
7- seek to (4 * **secondEmptyPlace** * (2* m +1)) , read integer and assign it to **thirdEmptyPlace**
8- seek to (4 * **thirdEmptyPlace** * (2* m +1)) , read integer and assign it to **nextEmptyPlace**
9- clear all nodes of the page, add **leftRecords** to it
10- seek to (**pageIndex** * 4 * (2 * m + 1)) , and write the page at this offset
11- make **newPage**, add the **rightRecords** to it
12- seek to (**firstEmptyPlace** * 4 *(2*m + 1)) , and write **newPage** there
13- seek to (4), and write **nextEmptyPlace** there as the first empty place
14- set **parentPageIndex** = index of the parent page
15- go to the parent page, remove the node that represents the old page
16- add **middle** with reference to the current page
17- add **last** with reference = **firstEmptyPlace**
18- set **newMiddle** = the node at middle of the parent page nodes, **newLast** = the largest node
19- clear **leftRecords** , **rightRecords**
20- add all nodes <= **newMiddle** to **leftRecords**
21- add all nodes > **newMiddle** to **rightRecords**
22- make **newLeft**, add the **leftRecords** to it
23- make **newRight**, add the **rightRecords** to it
24- seek to (**secondEmptyPlace** * 4 * (2 * m +1)) , write the page **newLeft**
25- seek to (**thirdEmptyPlace** * 4 * (2* m +1)) , write the page **newRight**
26- clear all nodes of parent page
27- add **newMiddle** to parent page with reference = **secondEmptyPlace**
28- add **newLast** to parent page with reference = **thirdEmptyPlace**
29- sort the nodes of parent page
30- seek to (**parentPageIndex** * 4 * (2* m +1)) , write the parent page

e. Insert a new node at the root page, while it's leaf and complete

This situation is delegated to this function:

```
int insertAndMakePageItselfParent (RandomAccessFile file, Page oldPage,
Node node, int pageIndex)
```

which follows the following algorithms:

1- insert the node at the correct index at **oldPage**
2- seek to (4) and set **nextEmptyPlace** = first empty place
3- seek to (4 * **nextEmptyPlace** * (2* m +1)) , read integer and assign it to **secondNextEmptyPlace**
4- set **middle** = the node at middle of the **oldPage** nodes, set its reference = **nextEmptyPlace**
5- set **last** = the largest node of the **oldPage** nodes, set its reference = **secondNextEmptyPlace**
6- make **oldPage.leafIndicator** = 1
7- clear all nodes of the **oldPage**, add **middle** and **last** to it
8- seek to (**pageIndex** * 4 *(2*m + 1)) , and write **oldPage** there
9- add all nodes <= **middle** to **leftRecords**
10- add all nodes > **middle** to **rightRecords**
11- make **leftPage**, add the **leftRecords**
12- make **rightPage**, add the **rightRecords**
13- seek to (**nextEmptyPlace** * 4 * (2 * m +1)) , write the page **leftPage**
14- seek to (**secondNextEmptyPlace** * 4 * (2* m +1) +4) , read integer and assign it to **newEmptyPlace**
15- seek to (**secondNextEmptyPlace** * 4 * (2* m +1)) , write the page **rightPage**
16- seek to (4), and write **newEmptyPlace** there as the first empty place
17- if node.**Key** > middle.**Key**
  17.1- return secondNextEmptyPlace;
  17.2- else → return nextEmptyPlace;

3) `boolean DeleteRecordFromIndex (String filename, int RecordID)`

This function deletes the node whose key = RecordID, it should also remove all instances of that key in the parent nodes and update them with new keys according to the pages they belong to. The deletion in a B-tree has some situations which are listed below:

a. delete a node from a page, which exceeds minimum number of nodes
b. delete a node from a page, which has strictly the minimum number of nodes

Now let's express each situation separately.

a. delete a node from a page, which exceeds minimum number of nodes

This situation is delegated to this function:

```
void deleteFromPageAboveMinimum (RandomAccessFile file, Page oldPage, int
Key, int pageIndex)
```

which follows the following algorithms:

1- set **indexOfNodeToRemove** = index of node with Key at the **oldPage**
2- if **indexOfNodeToRemove** == -1 → stop
3- Else
  3.1- set **oldLargestKey** = Largest Key of **oldPage**
  3.2- remove the node at index **indexOfNodeToRemove**
  3.3- shift all nodes next to the removed node leftward
  3.4- if the removed node had largest key
    3.4.1- set **newKey** = Largest Key of **oldPage**
    3.4.2- update all instances of the removed key with **newKey**
  3.5- seek to (**pageIndex** * 4 * (2 * m + 1))
  3.6- write **oldPage** again in the file.

b. delete a node from a page, which has strictly the minimum number of nodes

This situation is delegated to this function:

```
void deleteFromPageEqualToMinimum (RandomAccessFile file, Page oldPage,
int Key, int pageIndex)
```

which follows the following algorithms:

1- set **oldLargestKey** = Largest Key of **oldPage**
2-set **parentPageIndex** = the index of parent page of oldPage in the file
3- seek to (**parentPageIndex** * 4 * (2 * m + 1))
4- read a page and assign it to **parentPage**
5- set **oldPageIndexInParentPage** = index of **oldPage** at the **parentPage**
6- if **oldPage** is at the leftmost of the **parentPage**
  6.1- set **siblingPageIndex** = index of the right sibling page in the parentPage
  6.2- seek to (**siblingPageIndex** * 4 * (2 * *m* + 1)), read a page, assign it to **rightSiblingPage**
  6.3- if **rightSiblingPage** is above minimum number of nodes
    6.3.1- ***remove the first node from the right sibling, insert it into the old page***
    6.3.2- delete the node which has Key from oldPage, shift the rest of nodes leftward
    6.3.3- seek to (**pageIndex** * 4 * (2 * *m* + 1)), write oldPage
  6.4- **Else** → so both pages should be merged, one of them is to be deleted
    6.4.1- add all nodes of rightSiblingPage to oldPage, then delete **rightSiblingPage**
    6.4.2- update the value of first empty place if necessary
  6.5- delete the node which has **Key** from **oldPage**, shift the rest of nodes leftward

7- else if **oldPage** has left hand side siblings
  7.1- set **leftSiblingIndex** = index of the left sibling page in the **parentPage**
  7.2- seek to (**leftSiblingIndex***4*(2*m+1)), read a page, assign it to **leftSiblingPage**
  7.3- if **leftSiblingPage** is above minimum number of nodes
    7.3.1- remove the largest key node in **leftSiblingPage**, add it to **oldPage**
    7.3.2- delete the node which has **Key** from **oldPage**, shift the rest of nodes leftward
  7.4- else if **leftSiblingPage** has minimum number of nodes
    7.4.1- add all nodes of **leftSiblingPage**
    7.4.2- set **leftSibling** = node with largest key at **leftSiblingPage**
    7.4.3- delete **leftSiblingPage** and its references
    7.4.4- delete the node which has **Key** from **oldPage**, shift the rest of nodes leftward
    7.4.5- update the value of first empty place if necessary

4) `void DisplayIndexFileContent (String filename)`

This function shows the content of the file record by record. This function is delegated to this function:

```
void DisplayFile (RandomAccessFile file, int m)
```

whose algorithm is:

```
1- seek to (0)
2- for i = 0 : numOfRecords - 1
   2.1- read integer as leafIndicator , print it
   2.2- for j = 0 : m - 1
      2.2.1- read integer as key, print it
      2.2.2- read integer as reference, print it
      2.2.3- j = j + 1
   2.3- i = i + 1
   2.4- get back to step 2.1
```

5) `int SearchARecord (String filename, int RecordID)`

This function traverses the tree in search for a node whose key = RecordID. This function is delegated to this function:

`int search(int RecordID, RandomAccessFile file)`
whose algorithm is:

1- seek to (4 * (2 * m + 1)), read a page , assign it to **page**
2- for i = 0 : m - 1
  2.1- read integer as **leafIndicator**
  2.2- read a **node** from the **page**
  2.2- if the key of **node** == **RecordID**
    2.2.1- if the page is leaf → return the reference of **node** , stop
    2.3.2- Else → seek to ((Reference of **node**) * 4 * (2 * m + 1))
      2.3.2.1- return search (**RecordID**, file).
  2.3- if the key of **node** > **RecordID**
    2.3.1- if the **page** is not leaf → seek to ((Reference of **node**) * 4 * (2 * m + 1))
      2.3.1.1- return search (**RecordID**, file).
    2.4- Else → break , return -1

B) Team members roles

Name	Roles
Menna Allah Tarek Farouk	- isFileComplete - complexInsertionHandler - readPage - insertAndSplitCurrentAndMakeNewParent
Mai Mostafa Abdelrahman	- insertAndMakeNewLevelOfParents - insertAtNonCompletePage - getParentNodeIndex - getPageIndex
Mohamed Ahmed Abdelnabey	- sortList - deleteFromPageEqualToMinimum - search - insertAndMakePageItselfParent - WritePage
Abdelrahman Mohamed Ahmed	Class Page.java , class Node.java Functions : - CreateIndexFileFile - replaceOldKey
Kareem Ahmed Eltemsah	- deleteAllPageReferences - deletePage - deleteFromPageAboveMinimum - DisplayFile

Chapter 2. Design and Implementation

1) Functions of Class Main.java:

1- main()

```
public static void main(String[] args) throws IOException {
            Main m = new Main();
            m.start();
}
```

2- start()

```
private void start() throws IOException {
      String filename = "B-TreeIndex.bin" ;
      CreateIndexFileFile(filename, 5, 4);
      DisplayIndexFileContent(filename);
      //Insert: 5, 3, 21, 9, 1, 13, 2,7,10
      System.out.println(InsertNewRecordAtIndex(filename, 5 , 100));
      System.out.println(InsertNewRecordAtIndex(filename, 3 , 101));
      System.out.println(InsertNewRecordAtIndex(filename, 21 , 102));
      System.out.println(InsertNewRecordAtIndex(filename, 9 , 103));
      System.out.println(InsertNewRecordAtIndex(filename, 1 , 104));
      System.out.println(InsertNewRecordAtIndex(filename, 13 , 105));
      System.out.println(InsertNewRecordAtIndex(filename, 2 , 106));
      System.out.println(InsertNewRecordAtIndex(filename, 7 , 107));
      System.out.println(InsertNewRecordAtIndex(filename, 10 , 108));
      DisplayIndexFileContent(filename);

      //Delete 10 and then show the index file content
      if(DeleteRecordFromIndex(filename, 10) == false) {
         System.out.println("The record with id = 10 , doesn't exist in index file");
      }
      else   System.out.println("Successfully deleted");
      DisplayIndexFileContent(filename);
      //Delete 10 and then show the index file content
      if(DeleteRecordFromIndex(filename, 10) == false) {
         System.out.println("The record with id = 10 , doesn't exist in index file");
      }
      else   System.out.println("Successfully deleted");
      DisplayIndexFileContent(filename);
      //Delete 21 and then show the index file content
      if(DeleteRecordFromIndex(filename, 21) == false) {
         System.out.println("The record with id = 21 , doesn't exist in index file");
      }
      else   System.out.println("Successfully deleted 21");
      DisplayIndexFileContent(filename);
}
```

3- CreateIndexFileFile

```
void CreateIndexFileFile(String filename, int numberOfRecords, int m) throws
IOException {
	this.m = m;
	File f= new File(filename);
	f.delete()  ;
	RandomAccessFile file = new RandomAccessFile(filename, "rw");
	file.seek(0);
	for (int i = 1; i <= numberOfRecords; i++) {
		file.writeInt(-1); // write the leaf indicator
		if (i != numberOfRecords)
			file.writeInt(i);
		else
			file.writeInt(-1);
		file.writeInt(-1);
		for (int j = 0; j < (m - 1); j++) {
			file.writeInt(-1);
			file.writeInt(-1);
		}
	}
	file.close();
}
```

4- isFileComplete

```
boolean isFileComplete(String filename) throws IOException {
	boolean isFull = true ; ;
	RandomAccessFile file = new RandomAccessFile(filename, "rw");
	file.seek(4*(2 * m + 1));
	for(int i = 1 ; i < m ; i++) {
		Page temp = readPage(file);
		if ((!temp.isComplete() && temp.leafIndicator == 0)  ||
		temp.leafIndicator == -1 ){
			isFull = false ;
			break ;
		}
	}
	file.close();
	return isFull ;
}
```

5- InsertNewRecordAtIndex

```
int InsertNewRecordAtIndex(String filename, int RecordID, int Reference) throws
IOException {
	// insert function should return -1 if there is no place to insert the record
	// or the index of the node where the new record is inserted
	// if the record was inserted successfully.
	int returnIndex = -1;
	if(isFileComplete(filename) == true) {
		return -1 ;
	}
	RandomAccessFile file = new RandomAccessFile(filename, "rw");
	file.seek(4);
	Node node = new Node(RecordID, Reference);
	int nextEmptyPlace = file.readInt();
	if (nextEmptyPlace == 1) {
		// Our tree is empty , this is the first node in the first page
		file.seek(4 * (2 * m + 1));
		Page oldPage = readPage(file);
		insertAtNonCompletePage(file, oldPage, node, 1);
		file.close();
		returnIndex = 1;
	} else if (nextEmptyPlace == 2) {
		// Our tree has just 1 page , this page is either fully occupied or not
		file.seek(4 * (2 * m + 1));
		Page oldPage = readPage(file);
		if (oldPage.indexOfEmptyNode() == -1) {
			// Fully occupied , we need to split it
			returnIndex = insertAndMakePageItselfParent(file, oldPage, node , 1);
		} else {
			// Still has some nodes to insert at , no need to split the page
			insertAtNonCompletePage(file, oldPage, node, 1);
			returnIndex = 1;
		}
	} else {
		// Our tree has more than one page , we should select the appropriate one
		returnIndex = complexInsertionHandler(file, node);
	}
	file.close();
	return returnIndex;
}
```

6- writePage

```
void WritePage(RandomAccessFile file, Page page) throws IOException {
	file.writeInt(page.leafIndicator);
	for (int i = 0; i < m; i++) {
		file.writeInt(page.records.get(i).Key);
		file.writeInt(page.records.get(i).Reference);
	}
}
```

7- insertAtNonCompletePage

```
int insertAtNonCompletePage(RandomAccessFile file, Page page, Node node, int pageIndex)
throws IOException {
	if (page.leafIndicator == -1) {
		// This means the page is empty, we insert its first node
		// we should update its leaf indicator
		int nextEmptyPlace = -999;
		page.leafIndicator = 0;
		nextEmptyPlace = page.records.get(0).Key;
		page.records.get(0).Equals(new Node(-1, -1));
		file.seek(4);
		if (file.readInt() == pageIndex) {
		//This means the current page was the first empty page
		// we should update that to the page after it.
			file.seek(4);
			file.writeInt(nextEmptyPlace);
		}
	}
	int correctIndex = page.getCorrectIndexOfNewNode(node);
	int oldLargestKey = page.getLargestKey();
	if(correctIndex == -2) {
		page.records.add(node);
	}
	else   page.records.add(correctIndex, node);
	file.seek(4 * pageIndex * (2 * m + 1));
	WritePage(file, page);
	if(oldLargestKey < node.Key) {
		replaceOldKey(oldLargestKey, node.Key, file);
	}
	return pageIndex ;
}
```

8- readPage

```
Page readPage(RandomAccessFile file) throws IOException {
	int leafOrNotLeaf = file.readInt();
	ArrayList<Node> nodes = new ArrayList<Node>();
	for (int i = 0; i < m; i++) {
		Node temp = new Node(file.readInt(), file.readInt());
		nodes.add(temp);
	}
	Page page = new Page(m, leafOrNotLeaf, nodes);
	return page;
}
```

9- insertAndMakePageItselfParent

```
int insertAndMakePageItselfParent(RandomAccessFile file, Page oldPage, Node node ,
int pageIndex) throws IOException {
      // Choose this function in case you want to insert and split the page
      // but the page itself will be a parent

       int correctIndex = oldPage.getCorrectIndexOfNewNode(node);
       ArrayList<Node> records = new ArrayList<Node>();
       records.addAll(oldPage.records);
       if(correctIndex == -2) {
             records.add(node);
       }
       else   records.add(correctIndex, node);

       int currentPageIndex = pageIndex, nextEmptyPlace, secondNextEmptyPlace;
       file.seek(4);
       nextEmptyPlace = file.readInt();
       file.seek(nextEmptyPlace * (4 * (2 * m + 1)) + 4);
       secondNextEmptyPlace = file.readInt();

       int middleIndex = records.size() / 2 - 1;
       Node middleNode = new Node(), lastNode = new Node();
       middleNode.Equals(records.get(middleIndex));
       lastNode.Equals(records.get(records.size() - 1));
       middleNode.Reference = nextEmptyPlace;
       lastNode.Reference = secondNextEmptyPlace;

       oldPage.leafIndicator = 1;
       oldPage.records.clear();
       oldPage.records.add(middleNode);
       oldPage.records.add(lastNode);
       while (oldPage.records.size() < m) {
             oldPage.records.add(new Node(-1, -1));
       }
       file.seek(currentPageIndex * 4 * (2 * m + 1));
       WritePage(file, oldPage);

       ArrayList<Node> leftRecords = new ArrayList<Node>();
       ArrayList<Node> rightRecords = new ArrayList<Node>();
       for (int i = 0; i < records.size(); i++) {
             if (i <= middleIndex) {
                   leftRecords.add(records.get(i));
             } else {
                   rightRecords.add(records.get(i));
             }
       }
```

```
        Page leftPage = new Page(m, 0, leftRecords);
        Page rightPage = new Page(m, 0, rightRecords);

        while (leftPage.records.size() < m) {
                leftPage.records.add(new Node(-1, -1));
        }
        while (rightPage.records.size() < m) {
                rightPage.records.add(new Node(-1, -1));
        }

        file.seek(nextEmptyPlace * (4 * (2 * m + 1)));
        WritePage(file, leftPage);
        file.seek(secondNextEmptyPlace * (4 * (2 * m + 1)) + 4);
        int newEmptyPlace = file.readInt();
        file.seek(4);
        file.writeInt(newEmptyPlace);
        file.seek(secondNextEmptyPlace * 4 * (2 * m + 1));
        WritePage(file, rightPage);
        if (node.Key > middleNode.Key) {
                return secondNextEmptyPlace;
        } else {
                return nextEmptyPlace;
        }
}
```

10- complexInsertionHandler

```
int complexInsertionHandler(RandomAccessFile file, Node node) throws IOException {
        int pageIndex = getPageIndex(node.Key, file);
        int index = -1 ;
        file.seek(pageIndex*4*(2*m + 1));
        Page page = readPage(file);
        int parentPageIndex = getParentNodeIndex(page.getLargestKey(), pageIndex, file);
        file.seek(parentPageIndex*4*(2*m+1));
        Page parentPage = readPage(file);
        if(page.isComplete() ) {
                // This means it's a full page
                if(parentPage.isComplete()) {
                // even the parent page is full , we need to add new level of parents
                        return  insertAndMakeNewLevelOfParents(file, node, page, pageIndex,
parentPageIndex);
                }
                // we need to split only the current page
                // and insert a new largest key at parent node
                else   return insertAndSplitCurrentAndMakeNewParent(file, node, page,
pageIndex);
        }
        else {
        // just insert the new node at its page , no extra procedures needed
                index = insertAtNonCompletePage(file, page, node, pageIndex);
        }
        return index;
}
```

11- insertAndMakeNewLevelOfParents

```
int insertAndMakeNewLevelOfParents(RandomAccessFile file, Node node, Page page, int
pageIndex, int parentPageIndex) throws IOException {
	int correctIndex = page.getCorrectIndexOfNewNode(node);
	int oldLargestKey = page.getLargestKey();
	ArrayList<Node> records = new ArrayList<Node>();
	records.addAll(page.records);
	if(correctIndex == -2) {
		records.add(node);
	}
	else	records.add(correctIndex, node);
	file.seek(4);
	int firstEmptyPlace = file.readInt();
	file.seek(firstEmptyPlace*4*(2*m+1)+4);
	int secondEmptyPlace = file.readInt();
	file.seek(secondEmptyPlace*4*(2*m+1)+4);
	int thirdEmptyPlace = file.readInt();
	file.seek(thirdEmptyPlace*4*(2*m+1)+4);
	int nextEmptyPlace = file.readInt() ;
	sortList(records);
	int middleIndex = records.size() / 2 - 1;
	Node middleNode = new Node(), lastNode = new Node();
	middleNode.Equals(records.get(middleIndex));
	lastNode.Equals(records.get(records.size() - 1));

	ArrayList<Node> rightRecords = new ArrayList<Node>();
	ArrayList<Node> leftRecords = new ArrayList<Node>();
	for(int i = 0 ; i < records.size() ; i++) {
		Node temp = records.get(i);
		if(i <= middleIndex) {
			leftRecords.add(temp);
		}
		else	rightRecords.add(temp);
	}
	while(leftRecords.size() < m)	leftRecords.add(new Node(-1,-1));
	while(rightRecords.size() < m)	rightRecords.add(new Node(-1,-1));
	page.records.clear();
	page.records.addAll(leftRecords);
	page.sort();
	file.seek(pageIndex*4*(2*m+1));
	WritePage(file, page);
	Page newPage = new Page(m, 0, rightRecords);
	newPage.sort();
	file.seek(firstEmptyPlace*4*(2*m+1));
	WritePage(file, newPage);

	file.seek(4);
	file.writeInt(nextEmptyPlace);

	file.seek(parentPageIndex*4*(2*m+1));
	Page ParentPage = readPage(file);
	ParentPage.updateKey(oldLargestKey, middleNode.Key);
```

```
        middleNode.Reference = pageIndex ;
        lastNode.Reference = firstEmptyPlace ;
        ParentPage.records.add(lastNode);

        records.clear();
        records.addAll(ParentPage.records);
        Node newMiddle = new Node() , newLast = new Node();
        sortList(records);

        middleIndex = records.size() / 2 - 1;
        newMiddle.Equals(records.get(middleIndex));
        newLast.Equals(records.get(records.size() - 1));

        rightRecords.clear();
        leftRecords.clear();
        for(int i = 0 ; i < records.size() ; i++) {
              Node temp = records.get(i);
              if(i <= middleIndex) {
                    leftRecords.add(temp);
              }
              else   rightRecords.add(temp);
        }
        while(leftRecords.size() < m)    leftRecords.add(new Node(-1,-1));
        while(rightRecords.size() < m)   rightRecords.add(new Node(-1,-1));

        Page newLeft = new Page(m,1,leftRecords);
        Page newRight = new Page(m,1,rightRecords);
        newMiddle.Reference = secondEmptyPlace ;
        newLast.Reference = thirdEmptyPlace ;
        newLeft.sort();
        newRight.sort();
        file.seek(secondEmptyPlace*4*(2*m+1));
        WritePage(file, newLeft);
        file.seek(thirdEmptyPlace*4*(2*m+1));
        WritePage(file, newRight);

        ParentPage.records.clear();
        ParentPage.records.add(newMiddle);
        ParentPage.records.add(newLast);
        while(ParentPage.records.size() < m) {
              ParentPage.records.add(new Node(-1,-1));
        }
        ParentPage.sort();
        file.seek(parentPageIndex*4*(2*m+1));
        WritePage(file, ParentPage);


        file.seek(4*(2*m+1));
        return getPageIndex(node.Key, file);
}
```

12- insertAndSplitCurrentAndMakeNewParent

```
int insertAndSplitCurrentAndMakeNewParent(RandomAccessFile file, Node node , Page oldPage
, int pageIndex) throws IOException  {
	int correctIndex = oldPage.getCorrectIndexOfNewNode(node);
	int oldLargestKey = oldPage.getLargestKey();
	ArrayList<Node> records = new ArrayList<Node>();
	records.addAll(oldPage.records);
	if(correctIndex == -2) {
	// This means the key is larger than all the keys , so it should be inserted at the
end of the page
		records.add(node);
	}
	else   records.add(correctIndex, node);
	file.seek(4);
	int nextEmptyPlace = file.readInt();
	file.seek(nextEmptyPlace*4*(2*m+1)+4);
	int newEmptyPlace = file.readInt();
	int middleIndex = records.size() / 2 - 1;
	Node middleNode = new Node(), lastNode = new Node();
	middleNode.Equals(records.get(middleIndex));
	lastNode.Equals(records.get(records.size() - 1));

	ArrayList<Node> rightRecords = new ArrayList<Node>();
	ArrayList<Node> leftRecords = new ArrayList<Node>();
	for(int i = 0 ; i < records.size() ; i++) {
		Node temp = records.get(i);
		if(i <= middleIndex) {
			leftRecords.add(temp);
		}
		else   rightRecords.add(temp);
	}
	while(leftRecords.size() < m)    leftRecords.add(new Node(-1,-1));
	while(rightRecords.size() < m)   rightRecords.add(new Node(-1,-1));
	oldPage.records.clear();
	oldPage.records.addAll(leftRecords);
	file.seek(pageIndex*4*(2*m+1));
	WritePage(file, oldPage);
	Page newPage = new Page(m, 0, rightRecords);
	file.seek(nextEmptyPlace*4*(2*m+1));
	WritePage(file, newPage);
	file.seek(4);
	file.writeInt(newEmptyPlace);
	int parentIndex = getParentNodeIndex(oldLargestKey, pageIndex, file);
	file.seek(parentIndex*4*(2*m+1));
	Page ParentPage = readPage(file);
	ParentPage.updateKey(oldLargestKey, middleNode.Key);
	lastNode.Reference = nextEmptyPlace ;
	insertAtNonCompletePage(file, ParentPage, lastNode, parentIndex);
	file.seek(4*(2*m+1));
	return getPageIndex(node.Key, file);
}
```

13- DisplayIndexFileContent

```
void DisplayIndexFileContent(String filename) throws  IOException {
	// this method should display content of the file, each node in a line.
	DisplayFile(new RandomAccessFile(filename, "rw"), m);
}
```

14- DisplayFile

```
void DisplayFile(RandomAccessFile file, int m) throws IOException {
	this.m = m;
	file.seek(0);
	int numOfRecords = (int) (file.length() / (4 * (2 * m + 1)));

	System.out.println("_____________________________________________________________")
;
	for (int i = 0; i < numOfRecords; i++) {
		System.out.print((i) + ".\t" + file.readInt() + "\t");
		for (int j = 0; j < m; j++) {
			System.out.print("(" + file.readInt() + "\t" + file.readInt() +
")\t");
		}
		System.out.println();
	}
	file.close();
}
```

15- SearchARecord

```
int SearchARecord(String filename, int RecordID) throws IOException {
	// this method should return -1 if the record doesn't exist in the index
	// or the reference value to the data file if the record exist on the index

	RandomAccessFile file = new RandomAccessFile(filename, "rw");
	file.seek(4 * (2 * m + 1));
	int result = search(RecordID, file);
	file.close();
	return result;
}
```

16- search

```
int search(int RecordID, RandomAccessFile file) throws IOException
	Page page = readPage(file);
	for (int i = 0; i < m; i++) {
		if (page.records.get(i).Key == RecordID) {
			if (page.leafIndicator == 0) {
				return page.records.get(i).Reference;
			} else {
				file.seek(page.records.get(i).Reference * 4 * (2 * m + 1));
				return search(RecordID, file);
			}
		} else if (page.records.get(i).Key > RecordID) {
			if(page.leafIndicator == 1) {
				file.seek(page.records.get(i).Reference * 4 * (2 * m + 1));
				return search(RecordID, file);
			}
			else {
				break;
			}
		}
	}
	return -1;
}
```

17- getPageIndex

```
int getPageIndex (int key, RandomAccessFile file) throws IOException {
	file.seek(4*(2*m + 1));
	Page page = readPage(file);
	int idx = 0 , largestKey = page.getLargestKey();
	if( largestKey >= key ) {

		file.seek(4*(2*m + 1));
		search(key, file); // we call it to know which page it will stop at
		idx = (int) (file.getFilePointer()/(4*(2*m + 1))) - 1 ;
	}
	else {
		return getPageIndex(largestKey, file);
	}
	return idx;
}
```

18- replaceOldKey

```
void replaceOldKey(int oldKey , int newKey , RandomAccessFile file) throws
IOException {
		file.seek(4*(2*m + 1));
		for(int i = 0 ; i < m ; i++) {
			Page page = readPage(file);
			if(page.doIHaveThisNode(newKey) != null) {
				break ;
			}
			page.updateKey(oldKey, newKey);
			int seek = (int) (file.getFilePointer()- 4*(2*m + 1));
			file.seek(seek);
			WritePage(file, page);
		}
	}
```

19- getParentNodeIndex

```
int getParentNodeIndex(int key , int ref , RandomAccessFile file) throws IOException
{
	file.seek(4*(m*2 + 1));
	boolean isFound = false ;
	while(!isFound) {
		Page tempPage = readPage(file);
		if(tempPage.leafIndicator < 0) {
			break ;
		}
		else if(tempPage.leafIndicator == 1){
			Node tempNode = tempPage.doIHaveThisNode(key);
			if(tempNode != null) {
			   if(tempNode.Reference == ref) {
			      isFound = true ;
			      int idx = (int)((file.getFilePointer())/(4*(2*m + 1))) -1 ;
			      return idx ;
			   }
			   else {
			      file.seek(tempNode.Reference*4*(2*m+1));
			      continue;
			   }
			}

		}
	}
		return -1;
}
```

20- DeleteRecordFromIndex

```
boolean DeleteRecordFromIndex(String filename, int RecordID) throws IOException {
		boolean doesNodeExist = true ;
		if(SearchARecord(filename, RecordID) == -1) {
			// Then the record doesn't exist
			// You can't delete it
			return false ;
		}
		RandomAccessFile file = new RandomAccessFile(filename, "rw");
		int pageIndex = getPageIndex(RecordID, file);
		file.seek(pageIndex*4*(2*m+1));
		Page page = readPage(file);
		if(page.isAboveMinimumDescendants() == true) {
			deleteFromPageAboveMinimum(file, page, RecordID, pageIndex);
		}
		else if(page.isEqualToMinimumDescendants() == true) {
			deleteFromPageEqualToMinimum(file, page, RecordID, pageIndex);
		}

		file.close();
		return doesNodeExist ;
	}
```

21- deleteFromPageAboveMinimum

```
void deleteFromPageAboveMinimum(RandomAccessFile file, Page oldPage, int Key , int
pageIndex) throws IOException {
		int indexOfNodeToRemove = oldPage.indexOfNode(Key);
		if(indexOfNodeToRemove == -1)	return ;
		int oldLargestKey = oldPage.getLargestKey();
		oldPage.records.remove(indexOfNodeToRemove);
		if(oldLargestKey == Key) {
			int newKey = oldPage.getLargestKey();
			replaceOldKey(oldLargestKey, newKey, file);
		}
		while(oldPage.records.size() < m) {
			oldPage.records.add(new Node(-1,-1));
		}
		file.seek(pageIndex*4*(2*m+1));
		WritePage(file, oldPage);
	}
```

22- deleteFromPageEqualToMinimum

```
void deleteFromPageEqualToMinimum(RandomAccessFile file, Page oldPage, int Key , int
pageIndex) throws IOException  {
	int LargestKeyAtOldPage = oldPage.getLargestKey();
	int parentPageIndex = getParentNodeIndex(LargestKeyAtOldPage, pageIndex, file);
	file.seek(parentPageIndex*4*(2*m+1));
	Page parentPage = readPage(file);
	int oldPageIndexInParentPage = parentPage.indexOfNode(LargestKeyAtOldPage);
	Page rightSiblingPage = null , leftSiblingPage = null;
	if(oldPageIndexInParentPage == 0) {
	//The page is leftmost , if you intend to merge or borrow from a sibling page ,
the sibling should be at right only
		int siblingPageIndex = parentPage.records.get(1).Reference ;
		file.seek(siblingPageIndex*4*(2*m+1));
		rightSiblingPage = readPage(file);
		if(rightSiblingPage.isAboveMinimumDescendants()) {
		// Take a node from the sibling, insert it into the old page
			Node minimumNode = new Node();
			minimumNode.Equals(rightSiblingPage.records.get(0));
			deleteFromPageAboveMinimum(file,rightSiblingPage, minimumNode.Key,
siblingPageIndex);
			insertAtNonCompletePage(file, oldPage, minimumNode, pageIndex);
			deleteFromPageAboveMinimum(file, oldPage, Key, pageIndex);
			replaceOldKey(Key, oldPage.getLargestKey(), file);
		}
		else  {
			// There should be merge, delete one of them
			ArrayList<Node> siblingRecords = new ArrayList<Node> ();
			siblingRecords.addAll(rightSiblingPage.records);
			deletePage(file, siblingPageIndex);
			oldPage.records.addAll(siblingRecords);
			sortList(oldPage.records);
			deleteAllPageReferences(file, rightSiblingPage.getLargestKey());
			deleteFromPageAboveMinimum(file, oldPage, Key, pageIndex);
			file.seek(pageIndex*4*(2*m+1));
			WritePage(file, oldPage);
			replaceOldKey(Key, oldPage.getLargestKey(), file);
			file.seek(4);
			int firstEmptyPlace = file.readInt();
			if(firstEmptyPlace > siblingPageIndex) {
				file.seek(4);
				file.writeInt(siblingPageIndex);
				file.seek(siblingPageIndex*4*(2*m+1)+4);
				file.writeInt(firstEmptyPlace);
			}
```

```
                else if(firstEmptyPlace < siblingPageIndex&&firstEmptyPlace != 1){
                      file.seek(firstEmptyPlace*4*(2*m+1)+4);
                      int nextEmptyPlace = file.readInt();
                      file.seek(firstEmptyPlace*4*(2*m+1)+4);
                      file.writeInt(siblingPageIndex);
                      file.seek(siblingPageIndex*4*(2*m+1)+4);
                      file.writeInt(nextEmptyPlace);
                }
                else {
                      file.seek(4);
                      file.writeInt(siblingPageIndex);
                }

          }

    }
    else if(oldPageIndexInParentPage <= m-1) {
          // It has a L.H.S sibling
          int leftSiblingIndex =
          parentPage.records.get(oldPageIndexInParentPage-1).Reference;
          file.seek(leftSiblingIndex*4*(2*m+1));
          leftSiblingPage = readPage(file);
          //file.seek(rightSiblingIndex*4*(2*m+1));
          //rightSiblingPage = readPage(file);

          if(leftSiblingPage.isAboveMinimumDescendants()) {
                int siblingBorrowedNodeIndex =
          leftSiblingPage.indexOfNode(leftSiblingPage.getLargestKey());
                Node temp =
          leftSiblingPage.records.get(siblingBorrowedNodeIndex);
                oldPage.records.add(temp);
                            deleteFromPageAboveMinimum(file,
          leftSiblingPage,leftSiblingPage.getLargestKey() , leftSiblingIndex);
                deleteFromPageAboveMinimum(file, oldPage, Key, pageIndex);
                insertAtNonCompletePage(file, oldPage, temp, pageIndex);

          }
```

```
            else if(leftSiblingPage.isEqualToMinimumDescendants()){
                  for(int i = 0 ; i < leftSiblingPage.ActualSize();i++) {
                        insertAtNonCompletePage(file, oldPage,
                        leftSiblingPage.records.get(i), pageIndex);
                  }
                  int leftSiblingLargestKey = leftSiblingPage.getLargestKey();
                  deletePage(file, leftSiblingIndex);
                  deleteAllPageReferences(file, leftSiblingLargestKey);
                  deleteFromPageAboveMinimum(file, oldPage, Key, pageIndex);
                  file.seek(4);
                  int firstEmptyPlace = file.readInt();
                  if(firstEmptyPlace > leftSiblingIndex) {
                        file.seek(4);
                        file.writeInt(leftSiblingIndex);
                        file.seek(leftSiblingIndex*4*(2*m+1)+4);
                        file.writeInt(firstEmptyPlace);
                  }
                  else if(firstEmptyPlace < leftSiblingIndex&&firstEmptyPlace != 1){
                        file.seek(firstEmptyPlace*4*(2*m+1)+4);
                        int nextEmptyPlace = file.readInt();
                        file.seek(firstEmptyPlace*4*(2*m+1)+4);
                        file.writeInt(leftSiblingIndex);
                        file.seek(leftSiblingIndex*4*(2*m+1)+4);
                        file.writeInt(nextEmptyPlace);
                  }
                  else {
                        file.seek(4);
                        file.writeInt(leftSiblingIndex);
                  }
            }
      }
}
```

23- deleteAllPageReferences

```
void deleteAllPageReferences(RandomAccessFile file, int Key) throws IOException {
      file.seek(4*(2*m+1));
      int index = 1, nextIndex = -1 ;
      Page tempPage = readPage(file);
      while(tempPage.doIHaveThisNode(Key) != null) {
            nextIndex = tempPage.doIHaveThisNode(Key).Reference;
            deleteFromPageAboveMinimum(file, tempPage, Key, index);
            file.seek(nextIndex*4*(2*m+1));
            tempPage = readPage(file);
            index = nextIndex ;
      }

}
```

24- deletePage

```
void deletePage(RandomAccessFile file, int pageIndex) throws IOException {
		file.seek(pageIndex*4*(2*m+1));
		int leafIndicator = -1 ;
		ArrayList<Node> emptyRecords = new ArrayList<Node>();
		for(int i = 0; i < m; i++) {
			emptyRecords.add(new Node(-1,-1));
		}
		Page page = new Page(m, leafIndicator, emptyRecords);
		WritePage(file, page);

	}
```

25- sortList

```
public void sortList(ArrayList<Node> records) {
		Node temp = new Node();
		int n = records.size() ;
		for (int i = 1; i < n; i++) {
			for(int j = i ; j > 0 ; j--){
				if(records.get(j).Key < records.get(j-1).Key){
					temp.Equals(records.get(j));
					records.get(j).Equals( records.get(j-1));
					records.get(j-1).Equals(temp);
				}
			}
		}
		while(records.get(0).Key == -1) {
			records.remove(0);
			if(records.size() < m)	records.add(new Node(-1,-1));
		}
	}
```

2) Class Page.java:

```
public class Page {
	/*
	Each page here has a list of nodes. Each node has 2 integers , the first one
represents the record ID ,
	 the second one either represents the physical offset(if the page is leaf) ,
	 or represents the index of page to which this node points.
	*/
	int numberOfNodes ;
	int leafIndicator = 0 ; // if 0 -> leaf , if 1 -> non-leaf
	int currentCapacity = 0 ; // every time you add a node , increase this value
	ArrayList<Node> records = new ArrayList<Node>();

	public Page (int m) {
		numberOfNodes = m ;
	}

	public Page(int m , int leafIndicator , ArrayList<Node> nodes) {
		this.numberOfNodes = m ;
		this.leafIndicator = leafIndicator ;
		this.records = nodes ;
	}

	public int getLargestKey () {
		int max = -1 ;
		for(int i = 0 ; i < records.size() ; i++ ) {
			if(records.get(i).Key > max)	max = records.get(i).Key ;
		}
		return max;
	}

	public int indexOfEmptyNode () {
		for(int i = 0 ; i < records.size() ; i++) {
			if (records.get(i).Key == -1 && records.get(i).Reference == -1) {
				return i ;
			}
		}
		return -1;
	}

	public boolean addNode(Node node) {
		int i = indexOfEmptyNode() ;
		if(i != -1) {
			records.set(i, node);
			return true;
		}
		else {
			return false ;
		}
	}
```

```
	public int indexOfNode (int key) {
		for(int i = 0 ; i < records.size() ; i++) {
			if(records.get(i).Key == key)	return i ;
		}
		return -1 ;
	}

	public int getCorrectIndexOfNewNode (Node node) {
		int i = indexOfEmptyNode() ;
		for(int j = 0 ; j < records.size(); j++) {
			if(records.get(j).compareTo(node) > 0) {
				i = j ;
				break;
			}
		}
		if(i == -1 && node.Key > getLargestKey()) {
			i = -2;
		}
		return i ;
	}

	public void deleteEmptyNodeAt (int index) {
		records.remove(index);
	}

	public boolean isComplete() {
		return indexOfEmptyNode() == -1 ;
	}

	public void updateKey(int oldKey , int newKey) {
		if(leafIndicator == 0 || leafIndicator == -1) {
			return ;
		}
		for(int i = 0 ; i < records.size() ; i++) {
			if(records.get(i).Key == oldKey) {
				records.get(i).Key = newKey ;
				break;
			}
		}
	}

	public Node doIHaveThisNode(int key) {
		Node temp = null;
		for(int i = 0 ; i < records.size() ; i++) {
			if(records.get(i).Key == key) {
				temp = new Node() ;
				temp.Equals(records.get(i));
				break;
			}
		}
		return temp;
	}
```

```
	public boolean isAboveMinimumDescendants() {
		return ActualSize() > Math.ceil(records.size()/2) ;
	}

	public boolean isEqualToMinimumDescendants() {
		return ActualSize() == Math.ceil(records.size()/2.0) ;
	}

	public int ActualSize() {
		int count = 0 ;
		for(int i = 0 ; i < records.size() ; i++) {
			if(records.get(i).Key == -1 && records.get(i).Reference == -1)
	count++ ;
		}
		return records.size()-count ;
	}

	@Override
	public String toString () {
		return leafIndicator + " - " + records ;
	}

	public void sort() {
		Node temp = new Node();
		int n = records.size() ;
		for (int i = 1; i < n; i++) {
			for(int j = i ; j > 0 ; j--){
				if(records.get(j).Key < records.get(j-1).Key){
					temp.Equals(records.get(j));
					records.get(j).Equals( records.get(j-1));
					records.get(j-1).Equals(temp);
				}
			}
		}
		while(records.get(0).Key == -1) {
			records.remove(0);
			records.add(new Node(-1,-1));
		}
	}
}
```

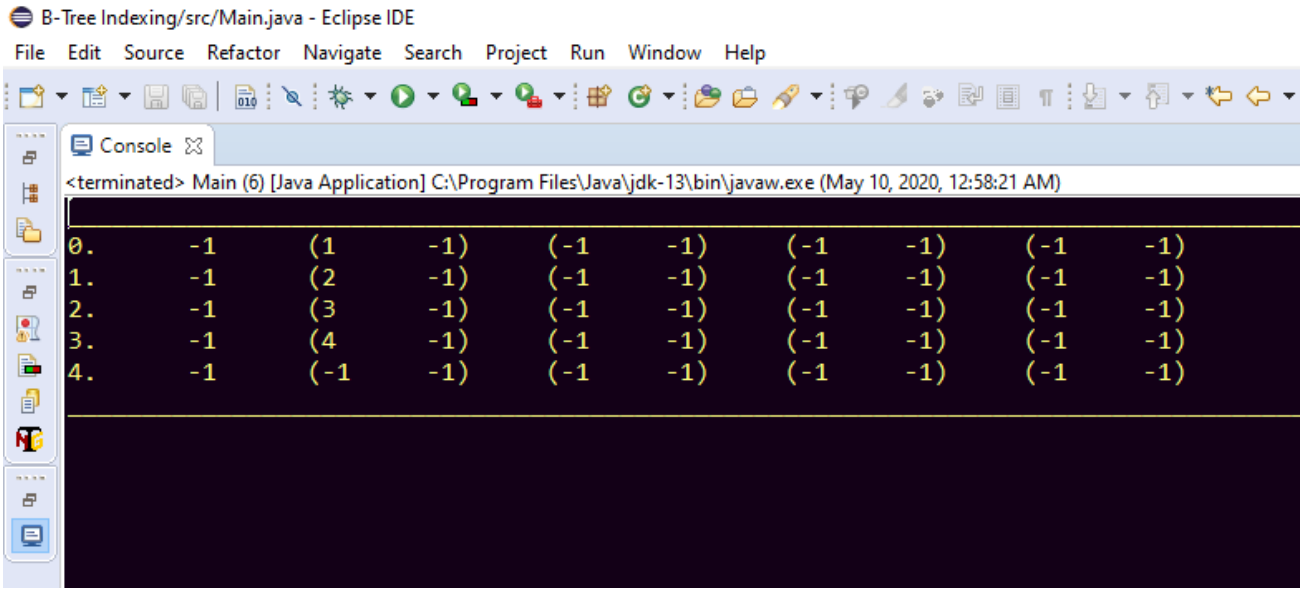
3) Class Node.java

```
import java.io.IOException;
import java.io.RandomAccessFile;

public class Node implements Comparable<Node>{
	public int Key , Reference ;

	public Node(int key, int offset) {
		Key = key ;
		Reference = offset ;
	}

	public Node() {}

	public void Equals (Node o) {
		this.Key = o.Key ;
		this.Reference = o.Reference ;
	}

	void WriteToFile(RandomAccessFile file) throws IOException {
		file.writeInt(Key);
		file.writeInt(Reference);
	}

	@Override
	public String toString () {
		return Key + " - " + Reference ;
	}

	@Override
	public int compareTo(Node o) {
		return Key - o.Key;
	}
}
```

Chapter 3. Test-Cases

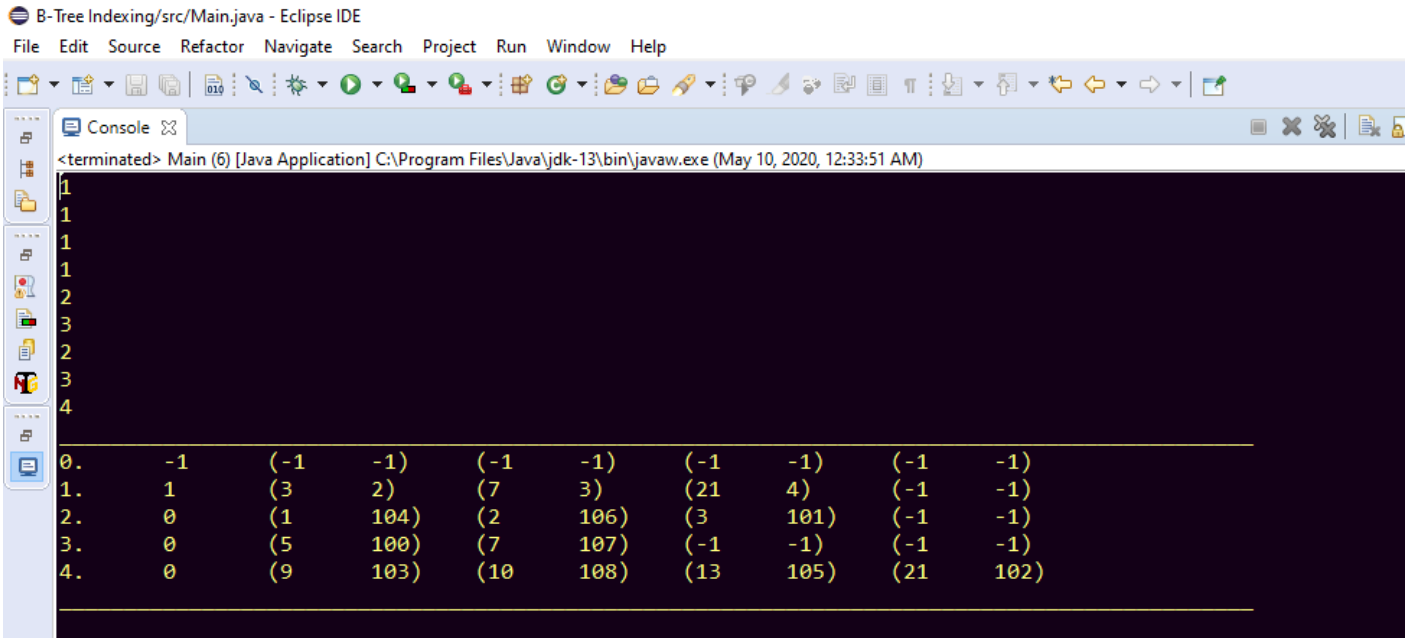
Test case 1 :
Create a tree with N = 5 and m = 4 and then show the index file content

The output shows the file contents after creation is done



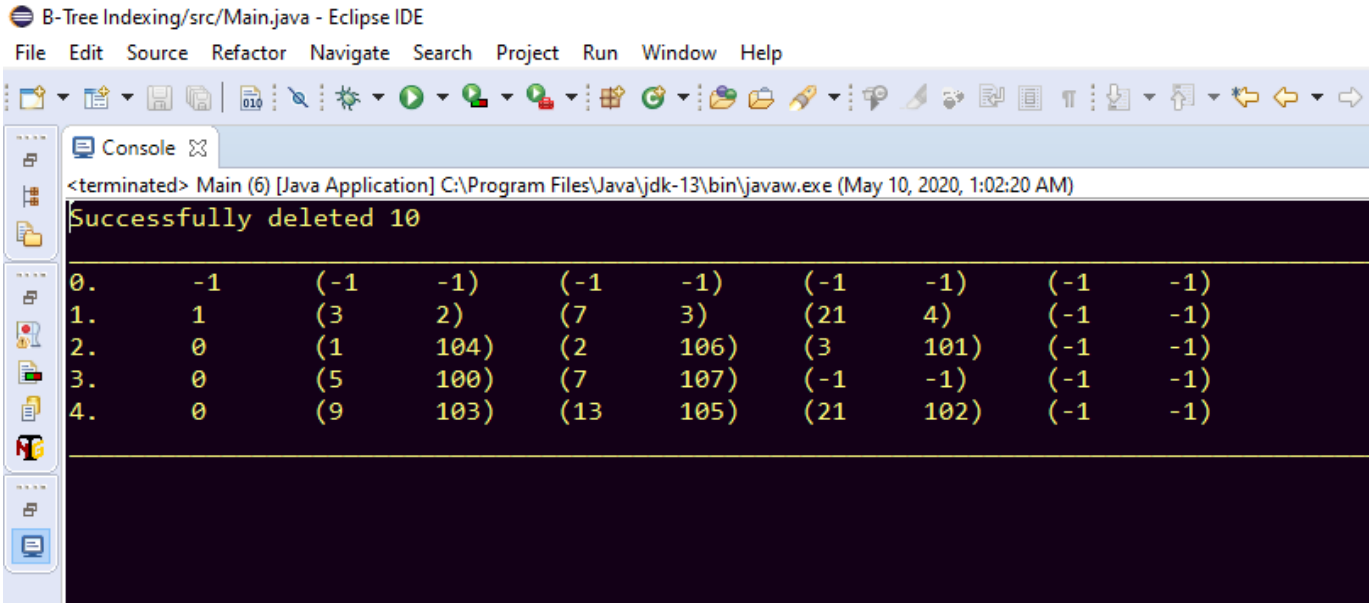
Test case 2 :
Insert: 5, 3, 21, 9, 1, 13, 2,7,10 and then show the index file content.

The output shows the index of page where each number was inserted, and finally the file contents after insertions are done



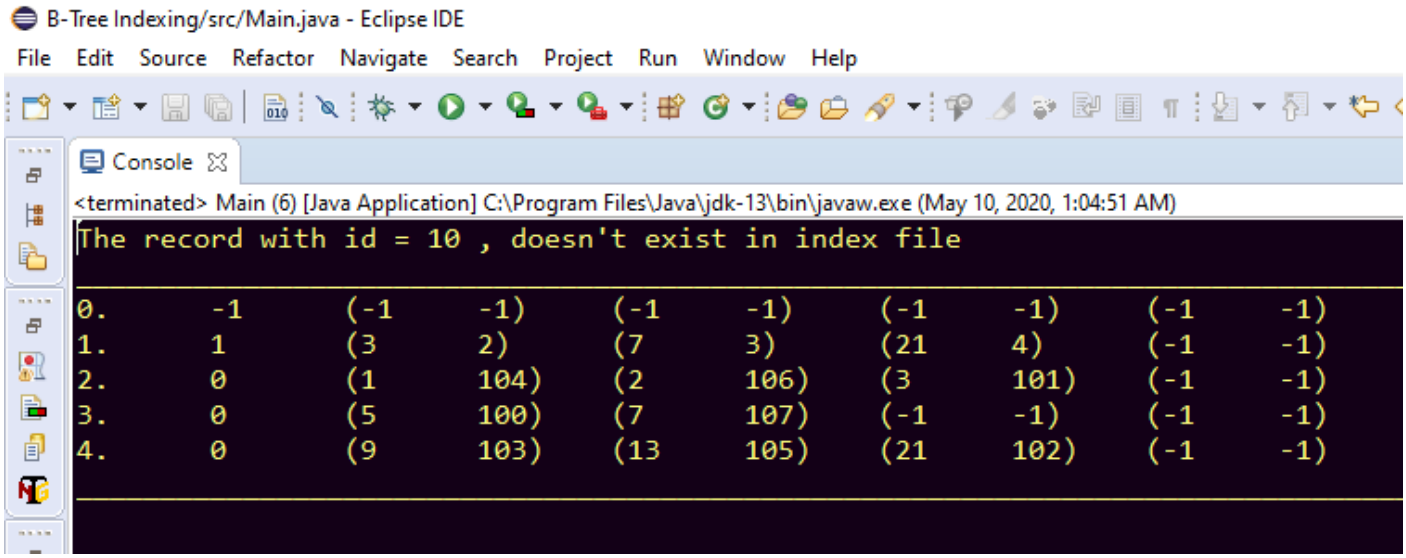
Test case 3 :
Delete 10 and then show the index file content

The output shows the file contents after deletion is done. The record exits so the deletion operation went successful



Test case 4 :
Delete 10 and then show the index file content

The output shows the file contents after deletion is done. The record doesn’t exist so the deletion operation failed



Test case 5 :
Delete 21 and then show the index file content

The output shows the file contents after deletion is done. The record exists so the deletion operation went successful