“Lab 3: B Tree and Indexing”

NAME: Sherif Mohamed Mostafa.

ID:20.

NAME: Ahmed Ali Abdelmeguid.

ID:5.

“Lab 3: B Tree and Indexing”

Problem statement:

The required was to implement a B-tree data structure and a simple search engine application utilizing this B-tree for indexing.

First: The B-tree:

The required is implementing a generic B-tree with each node stores a list of key value pairs and maintain the properties of the B-trees implementing the following main processes for the B-tree whilst maintaining its properties as stated in CLRS:

1. ***Insert:*** inserts a key value mapping into the tree.
2. ***Search:*** searches for the value of a given key.
3. ***Delete:*** deletes a certain key value mapping from the tree.

Second: Search Engine:

The required is to parse a set of Wikipedia documents in the XML format using Java DOM XML parser and maintain an index of these documents content using the B-Tree to be able to search them efficiently and that is done by implementing the following processes:

1. ***Index:*** index a set of documents given their file path to be able to search through them later.
2. ***Delete:*** delete a set of documents ID’s given their file path from the B-tree index if they were indexed before.
3. ***Search:*** search for a word or multiple words, for one word we return a list that contains the documents IDs that contains this search word, along with its rank, for multiple words search we return a list containing the document ID found in all search results of the words with the lowest rank found.

All these operations will be studied in more detail later for their complexities.

Implementation and design:

First: the B-tree:

The B-tree implementation consists of two classes:

- ***B-tree Node***:
  - This class contains all the data and processes concerning the B-tree node including number of keys, list of keys, list of corresponding values and list of children.
  - The node does not put any restrictions to its number of keys as it is done by the B-tree itself whilst inserting or deleting.
  - This class contains setters and getters for the data as specified by the given interface.
  - It also contains some additional methods that may be helpful.

Code Snippets:

```java
public class BtreeNode<K extends Comparable<K>, V> implements IBTreeNode<K, V> {

    private int numOfKeys;
    // Keys and values in this node index based from 0 to numberOfKeys-1
    private List<K> keys = new ArrayList<K>();
    private List<V> values = new ArrayList<V>();
    // An array list containing all the children of this node.
    private List<IBTreeNode<K, V>> children = new ArrayList<IBTreeNode<K, V>>();
    private boolean isLeaf;

    public BtreeNode() {// allocates a new node
        setNumOfKeys(0);
        setLeaf(true);
    }

    @Override
    public int getNumOfKeys() {
        return numOfKeys;
    }

    @Override
    public void setNumOfKeys(int numOfKeys) {
        this.numOfKeys = numOfKeys;
    }

    @Override
    public boolean isLeaf() {
        return isLeaf;
    }

    @Override
    public void setLeaf(boolean isLeaf) {
        this.isLeaf = isLeaf;
    }

    @Override
    public List<K> getKeys() {
        return keys;
    }
```

```java
@Override
public void setKeys(List<K> keys) {
    this.keys = keys;
    setNumOfKeys(keys.size());
}

@Override
public List<V> getValues() {
    return values;
}

@Override
public void setValues(List<V> values) {
    this.values = values;
}

@Override
public List<IBTreeNode<K, V>> getChildren() {
    return children;
}

@Override
public void setChildren(List<IBTreeNode<K, V>> children) {
    this.children = children;
}
```

Additional methods:

```java
public void addChild(IBTreeNode<K, V> child) { // Adds a child at the end of the list
    children.add(child);
}

public void addChild(IBTreeNode<K, V> child, int location) {// Adds a child at the specified location right shifting
                                                             // any children at higher locations.
    children.add(location, child);
}

public void removeChild(IBTreeNode<K, V> child) {// removes the given child node from the list possibly shifting all
                                                  // those children to its right to the left
    children.remove(child);
}

public void removeChild(int location) {// removes child node from the given location in the list possibly shifting
                                        // all those children to its right to the left
    children.remove(location);
}

// same with the keys/values lists:
public void addEntry(K key, V value) {
    numOfKeys++;
    keys.add(key);
    values.add(value);
}

public void addEntry(K key, V value, int location) {
    numOfKeys++;
    keys.add(location, key);
    values.add(location, value);
}

public void removeEntry(K key) {// given the key remove this entry.
    int entryLoc = keys.indexOf(key);
    keys.remove(entryLoc);
    values.remove(entryLoc);
    numOfKeys--;
}

public void removeEntryWithLocation(int location) {// given the location remove this entry.
    keys.remove(location);
    values.remove(location);
    numOfKeys--;
```

Testing method (recursive traversal):

```java
// Testing function to print out the sub tree rooted at this node by recursively
// traversing this subtree in order:
public void printInOrder() {// number of children = number of keys +1 unless this node is a leaf.
    int i;
    for (i = 0; i < getNumOfKeys(); i++) {// in order of this node and its first n children.
        if (!isLeaf())
            ((BtreeNode<K, V>) children.get(i)).printInOrder();
        System.out.print(" " + keys.get(i));
    }
    if (!isLeaf())
        ((BtreeNode<K, V>) children.get(i)).printInOrder();// for n+1 child.
}
```

- ***The B-tree class***:
  - The main class implementing the B-tree data structure.
  - It is pointer based having a reference to the root of the tree which is initialized at the first insertion operation and is the only node to be in the main memory all time and may be changed during execution.
  - NOTE: the implementation of the b tree with disk read and disk write is not required but if it were only the needed nodes and the root would be in memory.
  - The minimum degree of the b tree is given through the constructor.
  - The insert method inserts a new key value mapping into the tree and does nothing if the key already exists.
  - It initializes a new root in case the tree was empty.
  - Insert is done according to CLRS with handling all insert cases implementing : Insert, Insert non-full, split-child.
  - It basically recursively descends down the tree to get the correct leaf location for insertion but makes sure that the descend is always to a non-full node.
  - Search is done node by node starting from the root and recursing down the tree with n-way comparisons (n is number of keys in the node) for each node.
  - Delete is done according to CLRS implementing: delete, delete-leaf, delete non-leaf and merge.
  - It basically recursively descends down the tree to delete the key value mapping but makes sure that the descend is never to a node having minimum number of keys.
  - If the tree doesn’t contain the key deletion returns false.

Code Snippets:

```java
public class MyBtree<K extends Comparable<K>, V> implements IBTree<K, V> {// the main class implementing the B tree data
                                                                          // Structure.

    private int t;// the minimum degree of the B tree to be determined in the constructor.
    private BtreeNode<K, V> root;// to be set as a newly allocated node in the B tree in case the B tree was
                                 // allocated.

    public MyBtree() {// if t wasn't specified set it as the minimum possible degree 2.
        this.t = 2;
    }

    public MyBtree(int t) {
        this.t = t;
        if (t < 2)
            throw new RuntimeErrorException(null);
    }

    @Override
    public int getMinimumDegree() {
        return t;
    }

    public void setRoot(BtreeNode<K, V> root) {
        this.root = root;
    }

    public boolean isEmpty() {
        return root.getNumOfKeys() == 0;
    }

    @Override
    public IBTreeNode<K, V> getRoot() {
        return root;
    }
```

INSERT:

```java
private void insertNonFull(K key, V value, BtreeNode<K, V> node) {// a recursive function that assumes that the
                                                                  // recursion
    // descends to a non full node and goes on till
    // inserting in a leaf

    int i = node.getNumOfKeys() - 1;// last key location in this node.
    if (node.isLeaf()) {
        while (i >= 0 && node.getKeys().get(i).compareTo(key) > 0)
            i--;
        i++;
        node.addEntry(key, value, i);// add the new key value mapping possibly shifting other existing entries to
                                     // the right
    } else// not a leaf so recursively descend
    {
        // find correct child:
        while (i >= 0 && node.getKeys().get(i).compareTo(key) > 0)
            i--;
        i++;
        if (node.getChildren().get(i).getNumOfKeys() == 2 * getMinimumDegree() - 1) {// found child was full so need
                                                                                     // to split it before
                                                                                     // descend

            splitChild(i, node);
            if (key.compareTo(node.getKeys().get(i)) > 0)// should be placed in the second child
                i++;
        }
        insertNonFull(key, value, (BtreeNode<K, V>) node.getChildren().get(i));
    }
}
```

```java
@Override
public void insert(K key, V value) {
    if (root == null)
        root = new BtreeNode<>();
    if (key == null || value == null) {
        throw new RuntimeErrorException(null);
    }
    if (this.contains(key, root)) {// the tree already contains an entry with the given key so do nothing.
        return;
    }
    // first if the root is full we split the root:
    if (root.getNumOfKeys() == 2 * getMinimumDegree() - 1) {
        BtreeNode<K, V> newRoot = new BtreeNode<K, V>();
        // new root is made and the old root is its first child, this increases the
        // height of the tree by one from up
        newRoot.setLeaf(false);
        newRoot.addChild(root);
        splitChild(0, newRoot);
        // after splitting we must decide which of the two resulting children is the
        // correct insertion position.
        int i = 0;
        if (key.compareTo(newRoot.getKeys().get(0)) > 0)// key is in the second position.
            i = 1;
        insertNonFull(key, value, (BtreeNode<K, V>) newRoot.getChildren().get(i));
        setRoot(newRoot);
    } else
        insertNonFull(key, value, root);
}
```

```java
private void splitChild(int location, IBTreeNode<K, V> node) {// given the parent node and the child location it
                                                               // splits the child into two children and pushes the
                                                               // median key up to the parent
    IBTreeNode<K, V> child = node.getChildren().get(location);
    IBTreeNode<K, V> sibling = new BtreeNode<K, V>();
    sibling.setNumOfKeys(getMinimumDegree() - 1);
    sibling.setLeaf(child.isLeaf());
    // to remove the entries from the array list of the child, extra temporary array
    // lists are to be used to decrease the time complexity
    ArrayList<K> tempKeys = new ArrayList<K>();
    ArrayList<V> tempValues = new ArrayList<V>();
    ArrayList<IBTreeNode<K, V>> tempChildren = new ArrayList<IBTreeNode<K, V>>();
    // set the entries of the key as the last t entries in the node and also the
    // children if the node wasn't a leaf
    for (int i = 0; i < getMinimumDegree() - 1; i++) {
        tempKeys.add(child.getKeys().get(i));
        tempValues.add(child.getValues().get(i));
        sibling.getKeys().add(child.getKeys().get(i + getMinimumDegree()));
        sibling.getValues().add(child.getValues().get(i + getMinimumDegree()));
        if (!child.isLeaf()) {
            tempChildren.add(child.getChildren().get(i));
            sibling.getChildren().add(child.getChildren().get(i + getMinimumDegree()));
        }
    }
    if (!child.isLeaf()) {
        sibling.getChildren().add(child.getChildren().get(2 * getMinimumDegree() - 1));
        tempChildren.add(child.getChildren().get(getMinimumDegree() - 1));
    }
    node.getChildren().add(location + 1, sibling);
    node.getKeys().add(location, child.getKeys().get(getMinimumDegree() - 1));
    node.getValues().add(location, child.getValues().get(getMinimumDegree() - 1));
    node.setNumOfKeys(node.getNumOfKeys() + 1);
    child.setChildren(tempChildren);
    child.setKeys(tempKeys);
    child.setValues(tempValues);
    child.setNumOfKeys(getMinimumDegree() - 1);
}
```

SEARCH:

```java
@Override
public V search(K key) {// searches for the value corresponding to the given key and returns it.
                        // returns null if the key is not in the b tree.
    if (key == null) {
        throw new RuntimeErrorException(null);
    }
    return searchByNode(key, root);
}

private V searchByNode(K key, IBTreeNode<K, V> node) {// recursively search for the key in the subtree rooted by
                                                        // this node.
    int i = 0;
    while (i < node.getNumOfKeys() && key.compareTo(node.getKeys().get(i)) > 0)
        i++;// traverse the node keys till finding the range in which the required key
            // should exist.
    if (i < node.getNumOfKeys() && key.compareTo(node.getKeys().get(i)) == 0)// key was found in this node at this
                                                                              // location.
        return node.getValues().get(i);
    if (node.isLeaf())// key wasn't found in this subtree.
        return null;
    // recursively descend the search.
    return searchByNode(key, node.getChildren().get(i));
}
```

DELETION:

```java
// get a key value mapping from left sibling to the child
private void getFromPrevious(int location, IBTreeNode<K, V> node) {
    IBTreeNode<K, V> child = node.getChildren().get(location);
    IBTreeNode<K, V> sibling = node.getChildren().get(location - 1);
    // last from sibling goes to parent and entry at location location - 1 in the
    // node goes to the child
    child.getKeys().add(0, node.getKeys().get(location - 1));
    child.getValues().add(0, node.getValues().get(location - 1));
    if (!child.isLeaf())
        child.getChildren().add(0, sibling.getChildren().get(sibling.getNumOfKeys()));
    node.getKeys().set(location - 1, sibling.getKeys().get(sibling.getNumOfKeys() - 1));
    node.getValues().set(location - 1, sibling.getValues().get(sibling.getNumOfKeys() - 1));
    sibling.getKeys().remove(sibling.getNumOfKeys() - 1);
    sibling.getValues().remove(sibling.getNumOfKeys() - 1);
    if (!sibling.isLeaf())
        sibling.getChildren().remove(sibling.getNumOfKeys());
    child.setNumOfKeys(child.getNumOfKeys() + 1);
    sibling.setNumOfKeys(sibling.getNumOfKeys() - 1);
}

// get a key value mapping from right sibling to the child
private void getFromNext(int location, IBTreeNode<K, V> node) {
    IBTreeNode<K, V> child = node.getChildren().get(location);
    IBTreeNode<K, V> sibling = node.getChildren().get(location + 1);
    // Analogous to getting from previous sibling:
    child.getKeys().add(node.getKeys().get(location));
    child.getValues().add(node.getValues().get(location));
    if (!child.isLeaf())
        child.getChildren().add(sibling.getChildren().get(0));
    node.getKeys().set(location, sibling.getKeys().get(0));
    node.getValues().set(location, sibling.getValues().get(0));
    sibling.getKeys().remove(0);
    sibling.getValues().remove(0);
    if (!sibling.isLeaf())
        sibling.getChildren().remove(0);
    child.setNumOfKeys(child.getNumOfKeys() + 1);
    sibling.setNumOfKeys(sibling.getNumOfKeys() - 1);
}
```

```java
// to get predecessor node traverse all the way down from the nearest keys to
// your left
private IBTreeNode<K, V> predecessor(int location, IBTreeNode<K, V> node) {
    IBTreeNode<K, V> currentNode = node.getChildren().get(location);
    while (!currentNode.isLeaf()) {
        currentNode = currentNode.getChildren().get(currentNode.getNumOfKeys());
    }
    return currentNode;
}
// to get successor node traverse all the way down to the nearest key location
// from your right.

private IBTreeNode<K, V> successor(int location, IBTreeNode<K, V> node) {
    IBTreeNode<K, V> currentNode = node.getChildren().get(location + 1);
    while (!currentNode.isLeaf()) {
        currentNode = currentNode.getChildren().get(0);
    }
    return currentNode;
}

private void addKey(int location, IBTreeNode<K, V> node) {
    // check to borrow from previous or next child:
    if (location != 0 && node.getChildren().get(location - 1).getNumOfKeys() > getMinimumDegree() - 1)
        getFromPrevious(location, node);
    else if (location != node.getNumOfKeys()
            && node.getChildren().get(location + 1).getNumOfKeys() > getMinimumDegree() - 1)
        getFromNext(location, node);
    // otherwise merge the child with one of his siblings.
    else {
        if (location != node.getNumOfKeys())
            merge(location, node);
        else
            merge(location - 1, node);
    }
}
```

```java
private void deleteNonLeaf(int location, IBTreeNode<K, V> node) {
    // if previous child has more that t-1 keys work with predecessor
    if (node.getChildren().get(location).getNumOfKeys() > getMinimumDegree() - 1) {
        IBTreeNode<K, V> predecessor = predecessor(location, node);
        K predecessorKey = predecessor.getKeys().get(predecessor.getNumOfKeys() - 1);
        V predecessorValue = predecessor.getValues().get(predecessor.getNumOfKeys() - 1);
        node.getKeys().set(location, predecessorKey);
        node.getValues().set(location, predecessorValue);
        delete(predecessorKey, node.getChildren().get(location));
    }
    // else if next child has more than t-1 keys work with successor
    else if (node.getChildren().get(location + 1).getNumOfKeys() > getMinimumDegree() - 1) {
        IBTreeNode<K, V> successor = successor(location, node);
        K successorKey = successor.getKeys().get(0);
        V successorValue = successor.getValues().get(0);
        node.getKeys().set(location, successorKey);
        node.getValues().set(location, successorValue);
        delete(successorKey, node.getChildren().get(location + 1));

    }
    // else we have to merge
    else {
        K key = node.getKeys().get(location);
        merge(location, node);
        delete(key, node.getChildren().get(location));
    }
}
```

```java
private void delete(K key, IBTreeNode<K, V> node) {// a function that recursively descends to delete the key from
                                                    // the tree, as it does so it makes sure that the descend is to
                                                    // a node containing at least t keys
    // get key range location
    int i = 0;
    while (i < node.getNumOfKeys() && key.compareTo(node.getKeys().get(i)) > 0)
        i++;
    // case key is in the node:
    if (i < node.getNumOfKeys() && key.compareTo(node.getKeys().get(i)) == 0) {
        if (node.isLeaf())
            deleteLeaf(i, node);
        else
            deleteNonLeaf(i, node);
    } else {
        // if descend will be to a child with less than t keys we need to add a key to
        // this child
        boolean last = (i == node.getNumOfKeys());
        if (node.getChildren().get(i).getNumOfKeys() < getMinimumDegree())
            addKey(i, node);
        // in case of merging the number of keys decreases
        boolean merged = (i > node.getNumOfKeys());
        // now the recursive descend
        if (last && merged)
            delete(key, node.getChildren().get(i - 1));
        else
            delete(key, node.getChildren().get(i));
    }
}

private void deleteLeaf(int location, IBTreeNode<K, V> node) {
    node.getKeys().remove(location);
    node.getValues().remove(location);
    node.setNumOfKeys(node.getNumOfKeys() - 1);
}
```

```java
// merging children at location and location + 1.
private void merge(int location, IBTreeNode<K, V> node) {
    IBTreeNode<K, V> child = node.getChildren().get(location);
    IBTreeNode<K, V> sibling = node.getChildren().get(location + 1);
    // placing the median key
    child.getKeys().add(node.getKeys().get(location));
    child.getValues().add(node.getValues().get(location));
    for (int i = 0; i < sibling.getNumOfKeys(); i++) {
        child.getKeys().add(sibling.getKeys().get(i));
        child.getValues().add(sibling.getValues().get(i));
        if (!child.isLeaf())
            child.getChildren().add(sibling.getChildren().get(i));
    }
    if (!child.isLeaf())
        child.getChildren().add(sibling.getChildren().get(sibling.getNumOfKeys()));
    node.getKeys().remove(location);
    node.getValues().remove(location);
    node.getChildren().remove(location + 1);// removing the sibling node to undergo garbage collection
    child.setNumOfKeys(child.getNumOfKeys() + sibling.getNumOfKeys() + 1);
    node.setNumOfKeys(node.getNumOfKeys() - 1);
}

@Override
public boolean delete(K key) {
    if (key == null)
        throw new RuntimeErrorException(null);
    if (root == null || !contains(key, root))
        return false;
    delete(key, root);
    // if now root is empty we remove it
    if (root.getNumOfKeys() == 0) {
        if (root.isLeaf())
            root = null;// now tree is completely empty
        else
            setRoot((BtreeNode<K, V>) root.getChildren().get(0));// set root as its only child
    }
    return true;
}
```

Second: Search Engine:

The Search Engine implementation consists of two classes:

- ***Search Result***:
  - This class contains all the data concerning the word stored in B-tree node including document ID, rank.
  - This class contains setters and getters for the data as specified by the given interface.

Code Snippets:

```java
package eg.edu.alexu.csd.filestructure.btree.cs_5_20;

import eg.edu.alexu.csd.filestructure.btree.ISearchResult;

public class MySearchResult implements ISearchResult {
    private String id;
    private int rank;

    public MySearchResult() {

    }

    public MySearchResult(String id,int rank) {
        this.id = id;
        this.rank = rank;
    }

    @Override
    public String getId() {
        return id;
    }

    @Override
    public void setId(String id) {
        this.id = id;

    }

    @Override
    public int getRank() {
        return rank;
    }

    @Override
    public void setRank(int rank) {
        this.rank = rank;
    }

}
```

- ***The Search Engine class***:
  - The class contains a B-tree to insert words into it and deleting them from it, also to be used in searching for these words.
  - Every node in the B-Tree has a key and a value.
  - The key is the word itself and the value is a list of Search Result in which every index in this list contains the document ID the word appeared in and its rank in this document.
  - Firstly, we index a set of documents given their file path using DOM XML parser.
  - By storing the words of each document in an array to loop through we can insert them in the B-tree.
  - The indexing process is described as follows, using a Boolean to detect if the word is already stored in the B-tree or not.
  - If the word is stored in the B-tree, we search through its list if we found the document ID then we only increase its rank in this document.
  - If the document ID is not found in the list then we insert a new search Result containing the document ID and rank initiated by value of one.
  - If the word is not found in the B-Tree then we insert it with a new Search Result containing the document ID and rank initiated by value of one.
  - Also we can index a multiple files given their directory by looping through them and entering folders in that directory if exists, then we apply the indexing process as described.
  - The deletion process is described as follows, given a file path contains the documents to be deleted we can get the words using the DOM XML parser and storing the words in an array to loop through.
  - At First, we check if the word is stored in the B-tree or not.
  - Then, if not we absolutely do nothing.
  - If the word stored in the B-tree, we search through its list if we found the document ID to be deleted then we remove it from the list.
  - After that we check if the list is empty or not, if the list is empty we delete the word from the B-tree.
  - The searching process is described as follows.
  - We can search for one word or multiple words.

- Search for one word: if the word is stored in the B-tree we return its list which contains the documents IDs it appeared in and its rank in each document.
- If not we return a new empty list.
- Search for multiple words: we get the smallest search result list from the words to compare with the other lists to decrease the time as much as we can.
- The document ID found in all the lists is taken and added to the list to be returned with the lowest rank found.

<u>Code Snippets:</u>

<u>Indexing:</u>

```java
package eg.edu.alexu.csd.filestructure.btree.cs_5_20;

import java.io.File;

public class MySearchEngine implements ISearchEngine {

    int t = 50;

    public MySearchEngine(int t) {
        this.t = t;
    }

    // B-tree to apply insertion and deletion and searching
    private MyBtree<String, List<ISearchResult>> BTree = new MyBtree<>(t);

    @Override
    public void indexWebPage(String filePath) {// content is an array contains all the words in the document
        // using DOM XML parser, we can get the documents in the files and storing the
        // words in the array
        if (filePath == null || filePath.isEmpty() || filePath.replaceAll(" ", "") == "") {
            throw new RuntimeErrorException(null);
        }
        String[] content;
        try {
            DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
            DocumentBuilder db = dbf.newDocumentBuilder();
            Document doc = db.parse(new FileInputStream(new File(filePath)));
            NodeList allDocs = doc.getElementsByTagName("doc");
            for (int i = 0; i < allDocs.getLength(); i++) {
                Element element = (Element) allDocs.item(i);
                content = element.getTextContent().toLowerCase().replaceAll("\n\n", " ").replaceAll("\n", " ")
                        .replaceFirst(" ", "").split("\\s");

                indexDocs(content, element.getAttribute("id"));
            }
        } catch (ParserConfigurationException | SAXException | IOException e) {
            throw new RuntimeErrorException(null);
        }
    }
```

```java
private void indexDocs(String[] content, String DocID) {
    boolean foundWord = false;
    boolean foundDoc = false;
    for (int j = 0; j < content.length; j++) {
        foundDoc = false;
        if (BTree.getRoot()!=null)
            foundWord = BTree.contains(content[j], BTree.getRoot());// check if word is stored in the B-tree or not
        if (foundWord) {// word is stored
            List<ISearchResult> list = BTree.search(content[j]);
            for (int i = 0; i < list.size(); i++) {
                if (DocID.equals(list.get(i).getId())) {// check if document id is in the list or not, if found do
                                                       // the following
                    ISearchResult res = list.get(i);
                    int oldRank = res.getRank();
                    int newRank = oldRank + 1;
                    res.setRank(newRank);
                    foundDoc = true;
                    break;
                }
            }
            if (!foundDoc) {// if document id not found
                list.add(new MySearchResult(DocID, 1));
            }
        } else {// word is inserted for the first time
            List<ISearchResult> newList = new ArrayList<>();
            newList.add(new MySearchResult(DocID, 1));
            BTree.insert(content[j], newList);
        }
    }
}
```

```java
@Override
public void indexDirectory(String directoryPath) {
    if (directoryPath == null || directoryPath.isEmpty() || directoryPath.replaceAll(" ", "") == "") {
        throw new RuntimeErrorException(null);
    }
    File directory = new File(directoryPath);

    // Get all files from a directory.
    File[] fList = directory.listFiles();
    if (fList != null)
        for (File file : fList) {
            if (file.isFile()) {
                indexWebPage(file.getAbsolutePath());
            } else if (file.isDirectory()) {
                indexDirectory(file.getAbsolutePath());
            }
        }
}
```

Deletion:

```java
@Override
public void deleteWebPage(String filePath) {// using DOM XML parser to get the documents in file path and storing
                                            // the words in an array to apply deletion process
    if (filePath == null || filePath.isEmpty() || filePath.replaceAll(" ", "") == "") {
        throw new RuntimeErrorException(null);
    }
    String[] content;
    try {
        DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
        DocumentBuilder db = dbf.newDocumentBuilder();
        Document doc = db.parse(new FileInputStream(new File(filePath)));
        NodeList allDocs = doc.getElementsByTagName("doc");
        for (int i = 0; i < allDocs.getLength(); i++) {
            Element element = (Element) allDocs.item(i);
            content = element.getTextContent().toLowerCase().replaceAll("\n\n", " ").replaceAll("\n", " ")
                    .split(" ");

            deleteDocs(content, element.getAttribute("id"));
        }
    } catch (ParserConfigurationException | SAXException | IOException e) {
        throw new RuntimeErrorException(null);
    }
}
```

```java
private void deleteDocs(String[] content, String DocID) {
    for (int i = 0; i < content.length; i++) {
        if (content[i].length() != 0) {
            String word = content[i].toLowerCase();
            if (BTree.getRoot() == null || BTree.search(word) == null) { // this word appeared for first time
                continue;
            } else { // word is already present in btree (inserted before)
                List<ISearchResult> list = BTree.search(word);
                for (int j = 0; j < list.size(); j++) {
                    if (DocID.equals(list.get(j).getId())) {
                        list.remove(j);
                        if (list.size() == 0) {// if an empty list is created after deleting the document
                            BTree.delete(word);
                        }
                    }
                }
            }
        }
    }
}
```

Search:

```java
@Override
public List<ISearchResult> searchByWordWithRanking(String word) {
    if (word == null) {
        throw new RuntimeErrorException(null);
    }
    if (word.replaceAll("\\s+", "") == "" || BTree.contains(word, BTree.getRoot()) == false) {
        return new ArrayList<ISearchResult>();
    }
    word = word.replaceAll("\\s+", "");
    return BTree.search(word.toLowerCase());
}
```

```java
@Override
public List<ISearchResult> searchByMultipleWordWithRanking(String sentence) {
    if (sentence == null || sentence.isEmpty()) {
        throw new RuntimeErrorException(null);
    }
    if (sentence.replaceAll("\\s+", " ") == " ") {
        return new ArrayList<ISearchResult>();
    }
    List<ISearchResult> commonDocs = new ArrayList<>();
    List<List<ISearchResult>> wordsResults = new ArrayList<>();
    sentence = sentence.replaceAll("\\s+", " ");
    if (sentence.charAt(0) == ' ')
        sentence = sentence.replaceFirst(" ", "");
    String[] words = sentence.split("\\s");
    int foundID = 0;// counter of id if found in all list search results then we add it to the final
                    // list(commonDocs)
    for (int i = 0; i < words.length; i++) {
        if (BTree.getRoot() == null || BTree.search(words[i]) == null) { // this word appeared for first time
            continue;
        }
        List<ISearchResult> res = new ArrayList<>();
        res = searchByWordWithRanking(words[i]);
        if (res.size() != 0)
            wordsResults.add(res);
    }
    if (wordsResults.size() == 0)
        return new ArrayList<ISearchResult>();
    // using the smallest list to decrease the time as much as we can
    int SmallestListIndex = getSmallestList(wordsResults);
    List<ISearchResult> comparingList = wordsResults.get(SmallestListIndex);
    wordsResults.remove(SmallestListIndex);
    for (int i = 0; i < comparingList.size(); i++) {
        foundID = 0;
        for (int j = 0; j < wordsResults.size(); j++) {
            List<ISearchResult> temp = new ArrayList<>();
            temp = wordsResults.get(j);
            for (int k = 0; k < temp.size(); k++) {
                if (comparingList.get(i).getId().equals(temp.get(k).getId())) {
                    if (temp.get(k).getRank() < comparingList.get(i).getRank())
                        comparingList.get(i).setRank(temp.get(k).getRank());
                    foundID++;
                    break;
                }
            }
        }
        if (foundID == wordsResults.size())
            commonDocs.add(comparingList.get(i));
    }
    return commonDocs;
}
```

```java
private int getSmallestList(List<List<ISearchResult>> results) {
    if (results.size() == 0)
        return 0;
    int minSize = results.get(0).size();
    int index = 0;
    for (int i = 0; i < results.size(); i++) {
        if (results.get(i).size() < minSize) {
            minSize = results.get(i).size();
            index = i;
        }
    }
    return index;
}
```

Time and Space complexities:

First: The B-tree:

Let minimum degree be t and number of nodes in the tree be n.

- ***For the B-tree node***:
  - Time complexity:
    - All are setters and getters which include pointers and values adjustment so all are O(1).
  - Space complexity:
    - All the nodes have a maximum number of keys which is 2*t-1 and correspondingly maximum number of children 2*t and values 2*t-1.
    - So all the lists containing the key-values and the children pointers are O(t) storage for each node.
- ***For the B-tree itself***:

  Note that here we didn't implement the disk read and write and so we assume that the whole tree is to be allocated in memory at once.
  - Time complexity:
    - All setters and getters are O(1).
    - *Insert operation*: O(h) having h the height of the b-tree and as proven we have h is at most $\log_t((n+1)/2)$ so h is $O(\log_t n)$ and so inserting is $O(\log_t n)$ due to the recursive descend.
    - *Search operation*: as established for h and the number of keys of each node we have the search O(th) which is $O(t\log_t n)$ due to linear wise searching through each node down to the leaf. However it would be O(logn) in case of using binary search through each node which has negligible effect for small t compared to n.
    - *Deletion operation*: O(h) due to descend so similar to insertion it is $O(\log_t n)$.
  - Space complexity:
    - Space wise each B-tree object contains only a pointer to the root at any time and methods utilize the pointers connection.

- However as this implementation places the whole tree in the main memory we basically have the total space occupied by a tree be n*space occupied by each node so it is O(nt).
- However some study gives that a tree of min degree t has at most for keys $((2t)^{h+1}-1)/(2t-1)$ nodes each having 2t-1 keys maximum so a maximum of $((2t)^{h+1}-1)$ keys and space equals $((2t)^{h+1}-1)/(2t-1)$ * space occupied by each node which is 2*(2t-1) + 2t at most (O(t)) and having h at most $\log_t((n+1)/2)$ we get max number of keys as $(2^{\log_t((n+1)/2)}*t*(n+1) – 1)/(2t-1)$ with each key having a value mapping.

Note:

If CLRS implementation was done completely we would have added to the time complexities the access time (and what is calculated is CPU time) with access time as time of DISK READ and DISK WRITE operations as stated we would have access time for:

- *Insertion*: O(h) which is $O(\log_t n)$ as each descend O(1) DISK READ and WRITE occur.
- *Search*: O(h) too which is $O(\log_t n)$ but no DISK WRITE needed.
- *Delete*: O(h) which is $O(\log_t n)$ as each descend O(1) DISK READ and WRITE occur similar to insertion.

For space we have:

- Space complexity of each node is as it is however only the root is loaded into memory at all times.
- While performing an operation we have O(1) total nodes at a time which is at most 3 nodes (in deletion cases requiring the node, a child and a node for helping (merging and so..))
- So at most there is O(1) nodes in memory (4 including the root) and so the space storage in RAM is at most O(t) (as number of nodes became O(1)).
- In fact this is the main advantage of the b-tree as its size is not bounded by the RAM due to its access operations and keeping reference only for the current root.
- Note that any unneeded node should be de allocated from the RAM which is done automatically in case of the JAVA garbage collector.

Second: Search Engine:

Let n be the number of documents found in the XML file, m be the number of words in each document, k be the number of search results stored in the list of each word and t be number of words given in a sentence for multiple word search.

- ***For the Search Result***:
  - Time complexity:
    - All are setters and getters which include pointers and values adjustment so all are O(1).
  - Space complexity:
    - All are setters and getters which include pointers and values adjustment so all are O(1).
- ***For Search Engine***:
  - Time complexity:
    - *Indexing operation*: O(n*m*k) looping through all documents and storing the words of each document in an array and looping through them to insert in the B-tree and comparing the search results in the list of every word to find the document to be inserted.
    - *Deletion operation*: O(n*m*k) looping through all documents and storing the words of each document in an array and looping through them to delete from the B-tree and comparing the search results in the list of every word to find the document to be deleted.
    - *Search operation*: O(1) only returns the list of the search word in case of searching for one word.
    - In case of searching for multiple words, we need O(t) to get the search results lists of the given words, O(k) to find the smallest search result list found between words to be the comparing list, we need O((t-1)*k*k) temporary list to be compared, total of $O((t-1)*k^2)$.
    - The (t-1) part because we delete the smallest list from the words results list after finding it.
  - Space complexity:

- In indexing and deletion operations, we used three data structures in each; array contains the documents, array contains the words in each document and the list of search results for each word, that makes both operations $O(n*m*k)$.
- In searching for one word, we didn't use any data structures so $O(1)$.
- In case of searching for multiple words, we need $O(t*k)$ to store the search results lists of the given words, $O(k)$ for the smallest search result list found between words to be the comparing list, we need $t*O(k)$ temporary list to be compared, total of $O(t^2*k^2)$