REPORT

Lab 3

B Tree and Indexing

14: Toka Alaa Ahmed

52: Mennatulah Mohammed Ahmed

55: Nada Salama Mohammed

• The Problem Statement:

Introduction

1. B-Tree

B-trees are balanced search trees designed to work well on disks or other direct access secondary storage devices. Unlike the red-black trees, B-tree nodes can store multiple keys and have many children. If an internal B-tree node x contains x.n keys, then x has x.n + 1 children. The keys in node x serve as dividing points separating the range of keys handled by x into x.n + 1 sub-ranges, each handled by one child of x.

Simple Search Engine

You will be given a set of Wikipedia documents in the XML format and you are required to implement a simple search engine that given a search query of one or multiple words you should return the matched documents and order them based on the frequency of the query words in each wiki document, please check the requirements section for more details.

Requirements

B-Tree You are required to implement a generic B-Tree where each node stores key-value pairs and maintains the properties of the B-Trees.

- # code design for the simple search engine application

1) indexWebPage

Data structure used: hash map -> B-Tree

After parsing the xml file using Java DOM XML parser, for each document, we construct a word-rank map which is used to insert document words to the B-Tree with the document ID.

Code Snapshot:

```
@Override
public void indexWebPage(String filePath) {
    // TODO Auto-generated method stub
    if(filePath == null || filePath.length() == 0) {
        throw new RuntimeErrorException(null);
    }
    File fXmlFile = new File(filePath);
    if(fXmlFile.exists()) {
    try {
        DocumentBuilderFactory dbFactory = DocumentBuilderFactory.newInstance();
        DocumentBuilder dBuilder = dbFactory.newDocumentBuilder();
        Document doc = dBuilder.parse(fXmlFile);
        doc.getDocumentElement().normalize();
        NodeList nList = doc.getElementsByTagName("doc");
        for (int temp = 0; temp < nList.getLength(); temp++) {
            Node nNode = nList.item(temp);
            if (nNode.getNodeType() == Node.ELEMENT_NODE) {
                Element eElement = (Element) nNode;
                ID = eElement.getAttribute("id");
                insertDoc(nNode.getTextContent().toString());
            }

        }
        } catch (Exception e) {
        e.printStackTrace();
        }
}}
```

```
    private void insertDoc(String txt) {
        // TODO Auto-generated method stub
        String after = txt.trim().replaceAll(" +", " ");
        String text = after.replaceAll("\\r|\\n", " ");
        text = text.trim().replaceAll(" +", " ");
        text = text.toLowerCase();
        String[] splited = text.split("\\s+");
        Map< String,Integer> docMap = wordRankMap(splited);
        insertToBTree(docMap);
    }

    private Map< String,Integer> wordRankMap(String[] splited) {
        // TODO Auto-generated method stub
         Map< String,Integer> m =
                 new HashMap< String,Integer>();
        for(int i = 0; i< splited.length;i++) {
            String s = splited[i];
            if(m.containsKey(s)) {
                m.put(s, m.get(s)+1);
            }else {
                m.put(s, 1);
            }
        }
        return m;
    }

    private void insertToBTree(Map<String, Integer> docMap) {
        // TODO Auto-generated method stub
        for(Map.Entry< String,Integer> m: docMap.entrySet()) {
            ISearchResult searchResult = new SearchResult(ID, m.getValue());
            if(Tree.search(m.getKey()) == null) {
                List<ISearchResult> list = new ArrayList<>();
                list.add(searchResult);
                Tree.insert(m.getKey(), list);
            }else {
                Tree.search(m.getKey()).add(searchResult);
            }
        }
```

2) indexDirectory

It calls index web page method for files contained in the folders within the given directory recursively.

Code Snapshot:

```
@Override
public void indexDirectory(String directoryPath) {
    // TODO Auto-generated method stub
    if(directoryPath == null || directoryPath.length() == 0) {
        throw new RuntimeErrorException(null);
    }
    File file = new File(directoryPath);
    if(file.exists()) {
    for (File childFile : file.listFiles()) {
        String f = childFile.toString();
        if (childFile.isDirectory()) {
            indexDirectory(f);
        } else {
            indexWebPage(f);
        }
    }
    }
}
```

3) deleteWebPage

Data structure used: hash set -> B-Tree

After parsing the xml file using Java DOM XML parser, for each document, we construct a words-set which contains unrepeated set of words exist in the document. The words-set is used to search the B-tree for the words and delete its corresponding ID if exist.

Code Snapshot:

```
@Override
public void deleteWebPage(String filePath) {
    // TODO Auto-generated method stub
    if(filePath == null || filePath.length() == 0) {
        throw new RuntimeErrorException(null);
    }
    File fXmlFile = new File(filePath);
    if(fXmlFile.exists()) {
    try {
        DocumentBuilderFactory dbFactory = DocumentBuilderFactory.newInstance();
        DocumentBuilder dBuilder = dbFactory.newDocumentBuilder();
        Document doc = dBuilder.parse(fXmlFile);
        doc.getDocumentElement().normalize();
        NodeList nList = doc.getElementsByTagName("doc");
        for (int temp = 0; temp < nList.getLength(); temp++) {
            Node nNode = nList.item(temp);
            if (nNode.getNodeType() == Node.ELEMENT_NODE) {
                Element eElement = (Element) nNode;
                ID = eElement.getAttribute("id");
                deleteDoc(nNode.getTextContent().toString());
            }

        }
        } catch (Exception e) {
        e.printStackTrace();
        }}
}
private void deleteDoc(String txt) {
    // TODO Auto-generated method stub
    String after = txt.trim().replaceAll(" +", " ");
    String text = after.replaceAll("\\r|\\n", " ");
    text = text.trim().replaceAll(" +", " ");
    text = text.toLowerCase();
    String[] splited = text.split("\\s+");
    Set<String> hashSet = new HashSet<String>();
    hashSet.addAll(Arrays.asList(splited));
    deleteFromBtree(hashSet);
}
```

```
private void deleteFromBtree(Set<String> hashSet) {
    // TODO Auto-generated method stub
    boolean found = false;
    for(String s: hashSet) {
        List<ISearchResult> res = Tree.search(s);
        if(res != null) {
            for(int i = 0; i < res.size();i++) {
                if(res.get(i).getId().equals(ID)) {
                    Tree.search(s).remove(i);
                     found = true;
                     break;
                }
            }
            if(Tree.search(s).size() == 0) {
                Tree.delete(s);
            }
            if(!found) {// ID is not found
                break;
            }
        }else {//word is not found in the tree
            break;
        }
    }
}
```

4) searchByWordWithRanking

we use here B-Tree search which has values from indexing function. This function returns list of ISearch-Result that contains page ID and integer describes how many this word was repeated.

Code Snapshot:

```
@Override
public List<ISearchResult> searchByWordWithRanking(String word) {
    // TODO Auto-generated method stub
    if(word == null)
        throw new RuntimeErrorException(null);
    if(word == "")
        return new ArrayList<ISearchResult>();

    List<ISearchResult> list = Tree.search(word.toLowerCase());
    if(list == null) {
        return new ArrayList<ISearchResult>();
    }else return list;
}
```

5) searchByMultipleWordWithRanking

Data structure used: hash map -> B-Tree

This function get parameter as sentence so we split this sentence to separated words and begin to search or each one in the B-Tree.
The value of the first word is put in a list called result, the value of the second word is converted to map to be easy for comparing with the value of the first word. While looping thought the list we check if this ID is in the map or not. If No, delete this from result list. if Yes, we compare the rates and take the smaller one. We repeated this process for the rest words with the list result that comes from the last process.

code Snapshot:

```
@Override
public List<ISearchResult> searchByMultipleWordWithRanking(String sentence) {
    // TODO Auto-generated method stub
        if(sentence == null)
            throw new RuntimeErrorException(null);
         if(sentence == "")
                return new ArrayList<ISearchResult>();
         sentence = sentence.trim().replaceAll(" +", " ");
         if(sentence.charAt(0) == ' ') {
             sentence = sentence.substring(1, sentence.length());
         }
         String[] splited = (sentence.toLowerCase()).split("\\s+");
         List<ISearchResult> result = Tree.search(splited[0]);
         for(int i = 1; i < splited.length; i++) {
            List<ISearchResult> l = Tree.search(splited[i]);
            Map<String ,Integer> map = new HashMap<String, Integer>();
            while(!l.isEmpty()) {
                map.put(l.get(0).getId(), l.remove(0).getRank());
            }
            for(int j = 0; j < result.size(); j++) {
                if(map.containsKey(result.get(j).getId())){
                    if(map.get(result.get(j).getId()) < result.get(j).getRank()) {
                        result.get(j).setRank(map.get(result.get(j).getId()));
                    }
                } else {
                    result.remove(j);
                }
            }
         }
        return result;
}
```

- # The time and space complexity

1) B-Tree:

Time complexity:

O(t * h) for the insertion, deletion and search where t is the min degree and h is the height of the B-Tree.

Space of the B-Tree:

O(n*m) where n is the number of keys and the m is the space needed to store a value.

2) Search engine:

Time complexity:

1) Indexing a document:
   O(t*h*n) where t*h is the insertion time and n is the number of the unrepeated words in each document

2) deleting a document:
   O(t*h*n) where t*h is the deletion time and n is the number of the unrepeated words in each document.

Space of the B-Tree: O(n*m) where n is the number of words and the m is the space needed to store the list of IDs and Ranks.

Extra space is needed when constructing a word-rank map in indexing and set of words in deleting.