B-TREES IMPLEMENTATION

Zeyad Ahmed Ibrahim Zidan	19015709
Youssef Saber Saeed Mohammed	19016924
Aly Hassan Aly El-Shaarawy	19016013

Insertion

Problem Statement

It is required to construct a B-Tree using insertion. The insertion operation must maintain the properties of B-Tree.

Algorithm (Code Design)

We can solve this problem using 3 methods. Refer to t as the minimum degree.

1. **B_TREE_INSERT (K key, V value)**
    a. If (root.numOfKeys == 2t – 1)
        i. Node = allocate-node ().
        ii. Node.leaf = false.
        iii. Add root to Node children.
        iv. B_TREE_SPLIT (Node, 0).
        v. B_TREE_INSERT_NONFULL (Node, key, value).
        vi. This root = Node.
    b. Else B_TREE_INSERT_NONFULL (This root, key, value)
2. **B_TREE_SPLIT (Node node, int index)**
    a. rightChild = allocate-node ().
    b. leftChild = node.children[index].
    c. rightChild.leaf = leftChild.leaf.
    d. leftChild.keys.subList (t, 2t – 1) → rightChild.keys.
    e. Repeat for values and set the number of keys for rightChild.
    f. leftChild.keys.subList (t, 2t – 1).clear().
    g. Repeat for values and set the number of keys for leftChild.
    h. If (leftChild is not a leaf)
        i. leftChild.children.subList (t, 2t) → rightChild.children.
        ii. leftChild.children.subList (t, 2t).clear().
    i. Node.children.addAtIndex(index + 1, rightChild).

j. Move last element of leftChild to index position in node keys.
k. Set the number of keys for the node and leftChild.

3. **B_TREE_INSERT_NONFULL (Node node, K key, V value)**
    a. If (node is leaf)
        i. Insert the key and its value to node.
        ii. Sort the keys.
        iii. Maintain the number of keys for the node.
    b. Else
        i. Find the index of the next **child** where you would insert the key.
        ii. If (child.numOfKeys == 2t – 1)
            1. B_TREE_SPLIT (child, index)
            2. After split, check if the key can be inserted in the first child of split node.
            3. If so → increment index. If not → do nothing.
        iii. B_TREE_INSERT_NONFULL (node.children[index + 1], key, value).

Complexity Calculation

Time Complexity

Our implemented B-Tree costs a time complexity of ***O (log n)***.

Space Complexity

- A single node in a worst-case scenario would require 2n - 1 elements of keys, 2n - 1 elements of values, 2n elements of children that is an order of ***O (n)***.
- A single node in a best-case scenario would require at least n – 1 elements of keys, n – 1 elements of values, n elements of children that is an order of ***O (n)*** as well.
- That concludes that on average, a single node has a space complexity of ***O (n)***.
- For a tree having **n number of nodes** the tree space complexity would be ***O (n)***.

That sums up to a space complexity of $\boldsymbol{O(n^2)}$.

Deletion

Problem Statement

It is required to construct a B-Tree using Deletion. The Deletion operation must maintain the properties of B-Tree.

Algorithm (Code Design)

Our algorithm depends on mainly in recursion there are two function first

Function (delete) delete node only from leaf if leaf in initial it checks both successor and predecessor then choose the node have Number more than minimum degree and then call (rearrange) from the way back to root to make our tree balance

There are three cases in Re arrange

Case 1

Right child can donate to left child throw parent by gave parent from right child to parent and move from parent to right child and git less child from right child to left child.

Case 2

Same as case 1 but it left child donate instead of right child

Case 3

Same left child and right child have number of degrees less than minimum degree the first is add key from parent to left child and add right child to left child and remove right child from parent key then it have small case that parent have no keys then make left child our parent

Complexity Calculation

	Time	complexity
Delete	O(log n)	O(log n)

Search Engine

Problem Statement

Set of Wikipedia documents in the XML format and it is required to parse them and maintain an index of these documents content using the B-Tree to be able to search them efficiently.

Algorithm (Code Design)

For index web page function, we pass XML File and Map<ID,document> to parse this file then we split each document to string array of words with lower case (as required) then we insert each word as key to b-tree with corresponding value map<ID,frequency>.

```
indexWebPage(String filePath){
   parseXML(inputFile,indexedDoc);
   indexedDoc.forEach((id,doc) -> {
      splitted = splitStr(doc);
      for(s in splitted) {
                  wordMap = webSiteDocuments.search(s);
         if (!wordMap.have(id))
            wordMap.put(id, 1);
         else
            wordMap.add(id, wordMap.get(id) + 1);
      }
   });
}
```

For index directory function, we read all files in all directories and sub directories then insert them to b-tree using index web page function.

For read all files, we search through all sub directories using BFS and check for file to parse it or directory to push all its files to queue.

```
readAllFiles(String directoryPath, List<File> allFiles){
   directoryQueue.add(new File(directoryPath));
   while(!directoryQueue.isEmpty()){
      file = directoryQueue.poll();
      if(file.isDirectory()){
         if(file.listFiles() != null)
            directoryQueue.addAll(List.of(file.listFiles()));
      }
      else if(file.isFile()){
         allFiles.add(file);
      }
```

```
            }
        }
```

For delete web page function, we parse file to be deleted to Map<ID,document> then split each doc to string array of words with lower case (as required) then we search for every word and remove id then check id map corresponding to word is empty then we delete this word.

```
deleteWebPage(filePath) {
   parseXML(deleteFile,indexedDelete);
   indexedDelete.forEach((id,doc) -> {
      splitted = splitStr(doc);
      for(s in splitted){
         wordMap = webSiteDocuments.search(s.toLowerCase());
         if(wordMap != null && wordMap.containsKey(id)) {
            wordMap.remove(id);
            if(wordMap.size() == 0)
               webSiteDocuments.delete(s)
         }
      }
   });
}
```

For search by word with ranking, we search for word in b-tree then create list<Search Result> for each id and rank = frequency.

```
>searchByWordWithRanking(word) {
   wordMap = webSiteDocuments.search(word.toLowerCase());
   if(wordMap == null) return null;
   wordMap.forEach((id,freq) -> {
      SearchResult searchResult = new SearchResult(id.toString(),freq);
      wordRankedId.add(searchResult);
   });
   Return wordRankedId;
}
```

For search by multiple word with ranking, we split sentence to lower case words (as required) then we search for first word and loop on all other words and search for them, for each word we check id also in words id map, which contains all id for sentence, we filter id for each word that id must be in words id as well then we check for words id to remove id not in this word id so that words id contains id for the whole words of sentence.

```
searchByMultipleWordWithRanking(sentence) {
sentenceWords = splitStr(sentence.toLowerCase());
   wordsIdMap = webSiteDocuments.search(sentenceWords[0]);
   if(wordsIdMap == null) return null;

   for(i from 1to sentenceWords.length){
      nextWordMap = webSiteDocuments.search(sentenceWords[i]).clone();

      nextWordMap.clone().forEach((id,freq) -> {
         if(wordsIdMap.containsKey(id))
            wordsIdMap.put(id,Math.min(freq,wordsIdMap.get(id)));
         else
            nextWordMap.remove(id);
       });
      wordsIdMap.clone().forEach((id,rank) -> {
         if(!nextWordMap.containsKey(id))
            wordsIdMap.remove(id);
      });

   }
   wordsIdMap.forEach((id,rank) ->

          sentenceRankedID.add(new SearchResult(id.toString(),rank)));
   sentenceRankedID;
}
```

Complexity Calculation

Let w = number words in doc

Space: O(total number of words)

Index Web Page: Time: O(w * log w)

Index Directory: Time: O(w * log w)

Delete Web Page: Time: O(w * log w)

Search By Word with Ranking: Time: O(log w)

Search By Multiple Word with Ranking: Time: O(number words in sentence * log w)