Data Structures

HashSet is selected to store stop words. The objective of selecting a hash set is the contains() function with o(1) complexity. I simply determine whether the word is a stop word or not by using stopWords.Contains("wordToFind") function. You can find it in SearchEngine class named as "stopWordsSet".

Dictionary<String, Integer> is my choice to store file name as a key and counted word number in this file as a value. This data type is required in calculating frequency of the searched words in the related file. You can find it in SearchEngine class named as "files".

HashedDictionary<String, Alist<Integer>> is the data structure I mainly work on. It stores all the words we inserted from sports folder in the key part. The value part holds the occurences of the key in each file we work on. The main reason of why I used array list instead of a dictionary is the traverse type the array list has. From my point of view, maintaining the search speed is essential in a search motor. The array list holds file name as index number (I used the advantage of file names. They are all the numbers 001.txt … 511.txt etc. If the files were not named like that I would use dictionary type as a value in HashedDictionary and it would be more dynamical but slower) and counts of the key in files in those indexes. You can find it in SearchEngine class named as "hashedDict".

Java Code

RemoveStopWords

```
private int removeStopWords(String lineToClean, String fileName, int dirLength) {
    //for the texts named with numbers 001.txt etc.
    //returns number of non-stop word of the line
    int count= 0;
    for(String wordToCompare : lineToClean.split( regex: " ")){
        if(!stopWordsSet.contains(wordToCompare)) {

            wordToCompare= wordToCompare.replaceAll( regex: "[^a-zA-Z]", replacement: " ").replaceAll( regex: "\\s+", replacement: " ").trim();

            for(String word : wordToCompare.split( regex: " ")) {

                if (word.length() > 1) {
                    count++;
                    addToHashedDict(fileName, dirLength, word);
                }
            }

        }
    }
    return count;
}
```

O(1) complexity

Adds the words left to the hashedDictionary after the removal

SearchQuery

```
int i = 0;
while (valueIterator.hasNext()) { //traverses all the documents (texts in this project)
    int wordNumberOfFile = Integer.parseInt(valueIterator.next().toString());
    String nameOfFile = (String)  keyIterator.next();
    double freq = 0;

    int counter = 0; // to hold number of error while searching
    for(String word : query) { //to calculate total frequency
        AList<Integer> aListOfKey;
        try {
            aListOfKey = hashedDict.getValue(word); // returns array list of given word that stores occurrence count for each document
        }catch (Exception e) {
            continue;
        }
        if(aListOfKey==null){
            counter++;
            continue;
        }

        Object[] countValues = aListOfKey.toArray(); //holds word number in text

        try {
            freq += Double.parseDouble( countValues[i].toString() )  / wordNumberOfFile;
        }catch (ArrayIndexOutOfBoundsException e){ // means word does not include
            freq = 0;
        }catch (Exception e) {
            System.out.println("an error occured");
        }
    }
    if(counter == query.length)
        return "No documents found";

    if(maxFreq<=freq) {
        maxFreq = freq;
        mostRelatedFile = nameOfFile;
    }
    i++;
}
return mostRelatedFile;
```

Returns the most related file only. The relation is determined by calculating the frequencies and printing the file name that has the biggest frequency for the given word.

PERFORMANCE MATRIX

Load Factor	Hash Function	Collision Handling	Collision Count	Indexing Time(sec.)	Avg. Search Time(ns)	Min. Search Time	Max. Search Time
α=50%	SSF	LP	221	0.42	124.3	0	18300
		DH	252	0.41	144.4	0	20500
	PAF	LP	256	0.41	139.6	0	15100
		DH	265	0.42	104.3	0	16000
α=80%	SSF	LP	620	0.48	194	0	21500
		DH	714	0.47	243.2	0	19900
	PAF	LP	686	0.46	231.2	0	28700
		DH	728	0.49	139.8	0	19500

Load Factor

The more load factor causes the more collision. The more collision requires the more search & indexing time which is a huge disadvantage for any search motor. Nonetheless, we know that rehash is an expensive function for a program. So we have to consider an optimum load factor. In this test, 50% is better in collision handling, indexing time and search time.

Hash Function

In this graph, we see that there is no huge difference in SSF and PAF functions in avg. search time. Nonetheless, there are a little bit difference in collision number. We see that SSF had lesser collisions that is good for a search program.

Collision Handling Technique

Linear probing is a better choice in lesser collision count than double hashing. However, we cannot comment about the average search time. There is no big difference in search time.

Results (for 511txts and 1000 words to search)

1-PAF with double hashing is better than any combination in search operation.

2-Never select 80% if 50% is another option in load factor.

References:

-infotechgems.blogspot.com, “Java Collections – Performance (Time Complexity)”, Sunday, 6 November 2011 http://infotechgems.blogspot.com/2011/11/java-collections-performance-time.html

-Ali Cüvitoğlu, CME2201->Forum->Assignment-1 Questions