**3.0 Sorting the Tokens**

In this step,we firstly sort the Tokenizer output according to two orders. The first order is alphabetical sort of tokens and the second order is also according to alphabetical order of *filepath* string. For that sake, we have implemented compare function that performs comparisons according to the previous predicates order, using *Comparator* interface in Java.

|  |
| --- |
| class comparator implements Comparator <Pair>{  @Override  public int compare(Pair o1, Pair o2) {  // TODO Auto-generated method stub  if ( o1.getToken().compareTo(o2.getToken())<0)  return -1;   else if ( o1.getToken().compareTo(o2.getToken())>0)  return 1; else if( o1.getdocId().compareTo(o2.getdocId())<0)  return -1;  else if( o1.getdocId().compareTo(o2.getdocId())>0)  return 1;  return 0;  } } comparator c = new comparator(); list.sort(c); |

**4.0 posting list generation**

In this step, we iterate through the sorted list and gather the common tokens under one entry with their frequency counts and containing documents.  
For the indexer design, we preferred to use a data structure that follows the *HashMap* Key-Value mapping as well as maintains the insertion order of tokens list just like the *LinkedList structure*.   
*LinkedHashMap* combines these two advantages, it overcomes the artifact of *HashMap* which does not reserve the insertion order of keys, still it comprises Key-Value pairs.  
In our case, the Key-Value pair is basically a token string mapped to a *Posting* object respectively. The *Posting* object stores the document frequency and a *LinkedList* of document IDs. The document frequency does not account for redundant keyword inside single document, it just counts the token across the documents.

The code snippit below performs the tokens collapse:

|  |
| --- |
| **do** {  **if** (!ll.contains(list.get(j).getdocId())) {  entry.setValue(entry.getValue()+1);  ll.add(list.get(j).getdocId());  }  ++j; }**while**(entry.getKey().equals(list.get(j).getToken())); |

As for the time complexity analysis, the benefit of storing postings as *LinkedList* appears in the constant-time insertion cost O(1) when constructing the index.   
On the other hand, *LinkedList* has a worst-case of O(n) cost for sequential access during the document IDs retrieval. In this case, we justify that *ArrayList* is not the best alternative to resolve the access-insertion tradeoff since it increases the memory consumption. Even though *ArrayList* has random-access time, the main operation in this part is insertion.  
Finally, LinkedHashMap implements a doubly-linked list which means more pointers allocated in the memory. Since pointers stores memory addresses of elements in the linkedlist, we suggest to reduce memory allocation used to represent every address from 64-bit to 32-bit.

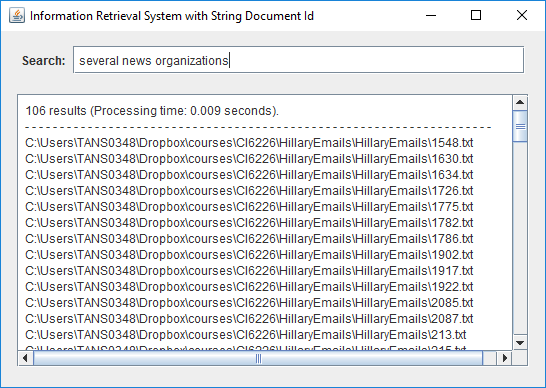
**5.0 Postings Lists Merging**

In this step, we implemented a basic merging algorithm for the intersection of the postings lists. The code snippet showing the implementation of the algorithm in Java is presented in Figure X. This algorithm takes two postings lists as its inputs, walks through the two postings lists in parallel, and returns the intersection of the lists. The complexity of the merging algorithm is O(m+n), where m and n are the number of entries in the two postings lists that are being merged.

We acknowledge that there are other more advanced algorithms for merging such as merging with a skip list to allow intersection to be processed in sublinear time instead of in time linear in the sizes of the postings lists. However, for this project, we did not invest additional time and effort in any of these algorithms because the size of the corpus is relatively small, so even with a basic merging algorithm, the system was able to process most if not all queries within reasonable time spans.

|  |
| --- |
| public static LinkedList<String> intersect(LinkedList<String> p1, LinkedList<String> p2) {  LinkedList<String> answer = new LinkedList<String>();  int step1 = 0;  int step2 = 0;  int n1 = p1.size();  int n2 = p2.size();  while(step1 < n1 && step2 < n2) {  String docId1 = p1.get(step1);  String docId2 = p2.get(step2);  int compareValue = docId1.compareTo(docId2);  if(compareValue == 0) { //docId1 = docId2  answer.add(docId1);  step1++;  step2++;  } else if(compareValue < 0) { //docId1 < docId2  step1++;  } else { //docId1 > docId2  step2++;  }  }  return answer;  } |

**6.0 The Information Retrieval System**



An information retrieval system was created with the simple interface shown in Figure X. The search function is triggered when a user presses <Enter> in the *Search* text field. The system then responds to the search by returning the followings:

* The paths to the files that fulfil the user’s search query, in the sense that the files contain all the terms that constitute the query. In other words, we considered all terms in the user’s query as joined by AND operators.
* Single-line summary that provides quick information about the number of files that fulfil the user’s query and the processing time of the query.

The time it took the system to create an index and the memory used to hold the index were measured. Having considered that the measured values might fluctuate slightly from one execution to another, we repeated the measurement over five executions (see Table X). The average time for indexing is 10.704 seconds whereas the average memory usage is 41171.109 kilobytes.

|  |  |  |
| --- | --- | --- |
| **Execution** | **Time (seconds)** | **Memory (kilobytes)** |
| 1 | 10.903 | 41171.273 |
| 2 | 10.571 | 41171.270 |
| 3 | 10.862 | 41171.234 |
| 4 | 10.760 | 41169.914 |
| 5 | 10.423 | 41171.852 |
| **Average** | **10.704** | **41171.109** |

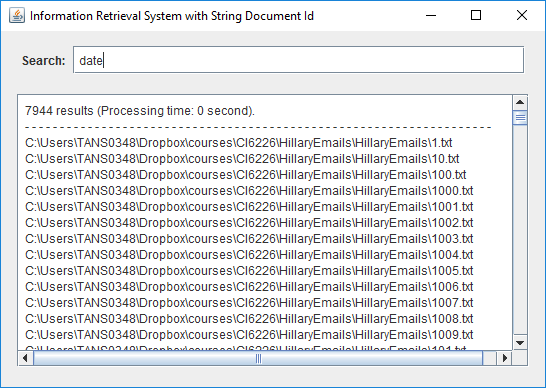
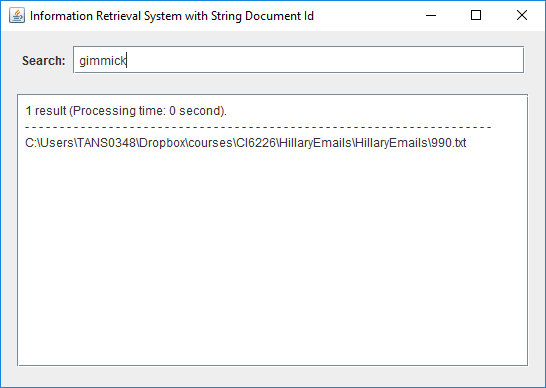
**6.1 Query Processing Procedure**

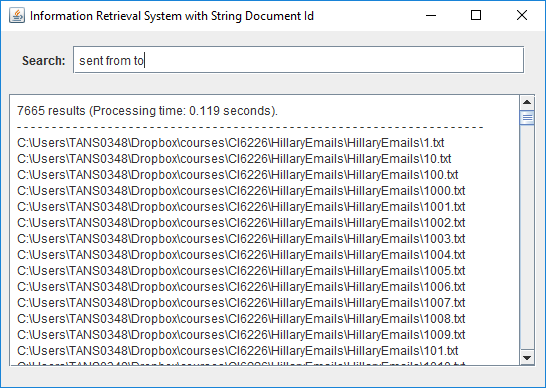
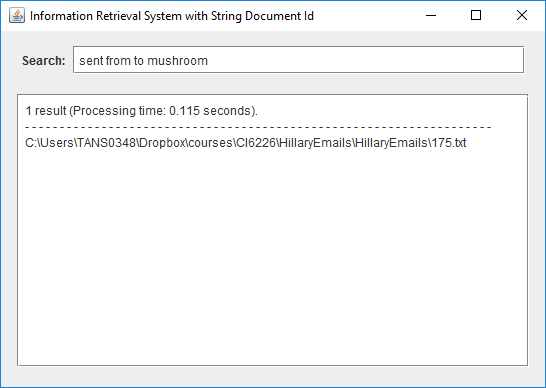
The processing of queries follows the procedure below:

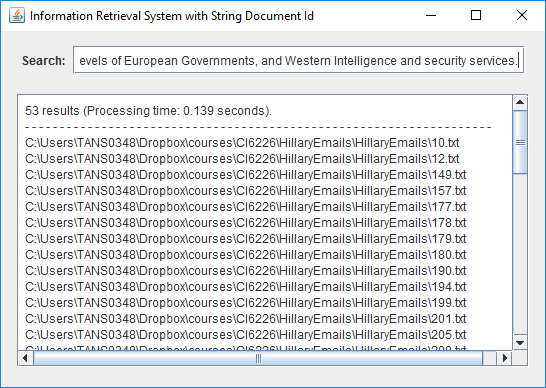
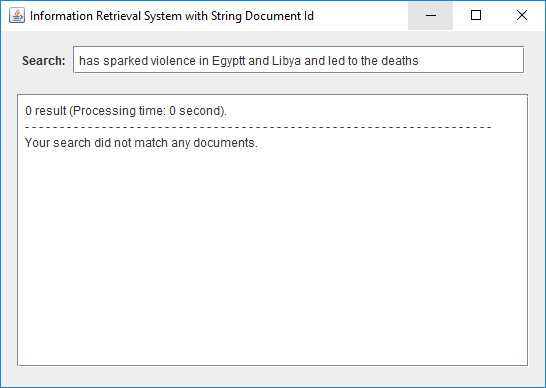
* Pass the query through the same tokenizer and linguistic module used for creating the index.
* For all modified tokens returned by the linguistic module, look them up in the index to get their postings lists. Note that if one of the words in the query does not exist in the index, our system would stop the look-up process immediately and return empty (or *null*) result so that no computation is wasted on completing the look-up and intersection steps.
* Get the intersection of these postings lists using the basic merging algorithm described in section 5.
* The entries (file paths) of the intersection are displayed as search results.

In the interest of learning how the system would behave in various scenarios, we put the system through a series of tests with queries that were chosen to test out different cases. Table X shows the queries used in the tests with the descriptions of test scenarios for which they were intended. In general, case 1 through 5 examine the relation between query processing time and query length, and they also serve to find out whether the document frequency of words in a query might have any effect on the processing time. Case 6 and 7 are chosen to verify whether the system is able to handle, in a decent manner, cases of non-existent or misspelled words and non-alphanumeric characters.

|  |  |  |
| --- | --- | --- |
| **Case No.** | **Query** | **Test Case Description** |
| 1 | date | Single word query of a common word. The document frequency of the word ‘date’ is 7944. |
| 2 | gimmick | Single word query of an uncommon word. The document frequency of the word ‘gimmick’ is 1. |
| 3 | sent from to | Multi-word query containing only common words. The document frequencies of ‘sent’, ‘from’, and ‘to’ are 7670, 7823, and 7857 respectively. |
| 4 | sent from to mushroom | Multi-word query containing common words and uncommon words. The document frequencies of ‘sent’, ‘from’, and ‘to’ are 7670, 7823, and 7857 respectively, whereas the document frequency of ‘mushroom’ is 1. |
| 5 | Sources with direct access to the Libyan National Transitional Council, as well as the highest levels of European Governments, and Western Intelligence and security services. | Exceptionally long query. |
| 6 | has sparked violence in **Egyptt** and Libya and led to the deaths | Query containing non-existent or misspelled words (i.e., Egyptt). |
| 7 | Case No. F-2015-04841 | Query containing non-alphanumeric characters. |

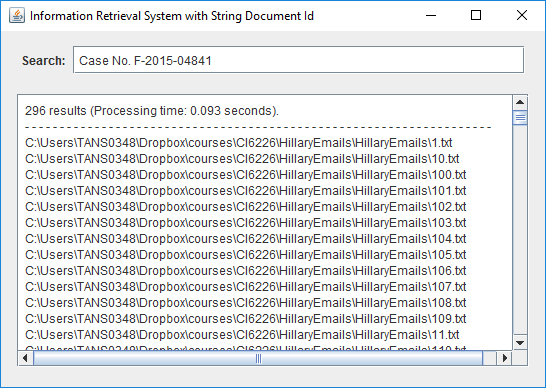


Figure X shows the search results and processing time for the seven cases described above. From the observations of the test outcomes, it seems the processing time largely depends on the length of queries. This is somehow expected because this system considers all queries as Boolean AND queries, the time spent on intersecting postings lists is thus positively correlated with the length of queries. In other words, the longer the queries, the more time would be spent on merging postings lists of the query words. Therefore, the processing time for single-word queries (case 1 and 2) is apparently smaller than the processing time for multi-word queries (case 3, 4, and 5). Whether the queries consist of common or uncommon words did not seem to affect the processing time in these cases. However, the query processing time for certain multi-word queries can be improved with a simple query optimization step that we will describe in the next section (section 6.2).

Case 6 has demonstrated that the system is able to handle non-existent or misspelled word efficiently. The processing time for case 6 is negligible because the system aborted the index look-up process and returned 0 result when the word ‘Egyptt’ was encountered in the query. In addition, case 7 has verified that queries containing non-alphanumeric characters are also properly handled by the system.

**6.2 Optimizing Query Processing with Document Frequencies**

To improve the efficiency of query processing, we added a sorting step to the query processing procedure described in section 6.1 so that the merging of postings lists is carried out in ascending order of document frequencies. Case 1 through 5 were tested again after this optimization step was incorporated into the system. Figure X shows the new search results and processing time for these five cases. This optimization step only affected the processing time for multi-word queries, while the processing time for single-word queries remained unchanged. The improvement was only prominent for case 4, in which the multi-word query contains an uncommon word ‘mushroom’ which has a document frequency of 1. In case 4, the processing time was reduced from 0.115 seconds to 0.002 seconds. However, in cases where all the words in the queries have similar document frequencies, the sorting step did not help the performance. In fact, in case 5, the query processing time increased slightly from 0.139 seconds to 0.148 seconds after the sorting step was introduced.

