Data Structures

Structure	Justification
ArrayList	Array- based list implementation provides fast append method which is useful when accumulating values to return (e.g. accumulating occurrences from multiple words when prefix searching). Can perform binary search if ordered which is often the case with occurrences, section numbers etc since they are added in ascending order when reading in the file. Slow remove and insert methods but this isn't important as these methods are seldom used; mostly used for storing information about words when preprocessing text and for accumulating search result to return (doesn't make sense to remove elements in pre-processed information and values are generally appended to list since text file is read sequentially).
HashMap	Array-based HashMap implementation allows for O(1) average-time O(n) worst case mapping between values which is useful when dealing with strings (e.g. mapping section titles to section indexes which makes searching with section constraints easier and faster). Can be used to implement other data helpful data structures. It is difficult to iterate through keys or values in a HashMap but this functionality is not really used since section constraints are always user specified.
HashSet	HashMap backed HashSet implementation allows for querying in O(1) of whether a set contains a specified element. This is useful in section constrained searches
HashPair	Used when grouping two elements together and provides sensible equals and hashing functionality required for above.
OccurrenceTrie	Can store information about a word at the node that terminates the word in the Trie. This is useful because it makes word searches O(word.length) instead of O(document.length). The preprocessing of the document to form a trie is justified because of this increased search efficiency and since the document will be loaded and searched multiple times. While similar advantages could be obtained from a HashMap search implementations, Tries also inherently store relations between search words (prefixes) useful for certain types of searches which is not present in the keys of HashMaps.
OccurrenceTrieNode	Required for the implementation of OccurrenceTrie. Stores a HashPair of lineNumber, columnNumber for each occurrence of the word in the document and the section numbers of these occurrences. These occurrences are stored in ascending order. This ordering stems naturally from reading through the document sequentially in preprocessing and is exploited in some search algorithms.

Data Structures from Preprocessing

Name	Data Structure	Contains	Generated
documentTrie	OccurenceTrie	The occurrences and section numbers of every word in the document	By reading through the document and traversing down trie, adding nodes, occurrences and section numbers as required
sectionIndexes	HashMap<String, HashPair<Integer, Integer>>	Mapping from section titles to the section number and its starting line number	By reading through the index file and creating an entry for each line
sectionStarts	ArrayList<Integer>	Starting line number of each section is at the index of the section number	While processing index file for sectionIndexes
stopWords	HashSet<String>	All stop words	Each line in stop words file is an element in the HashSet
textLines	HashMap<Integer, String>	Mapping from line number to content of line	While forming documentTrie, each line is added to HashMap. Could be stored on disc with references in main memory

Algorithms

Algorithm	Methodology	Justification
Binary Search	Comparison against middle value and repeating search for half of array	Provides O(log(n)) search, exploits ordering of occurrences inherent in document preprocessing
Intersection of ordered lists	Initialises a pointer for each array pointing to the first element. The values pointed at in each array are compared for intersection. If all are equal this value is appended to the result and all pointers are incremented and duplicates skipped. The pointer pointing to the minimum value is incremented. This is repeated until all pointers point past the end of their arrays.	Exploits inherent ordering of occurrences. Each list is traversed only once which significantly improves the possible O(numElementsPerList ^ numLists) complexity. Works for an arbitrary number of lists since it doesn't rely on nested for loops of a static depth
Union of ordered lists	Initialises a pointer for each array pointing to the first element. The minimum value pointed to is added to the result. Any pointer pointing to a value equal to this minimum is incremented to avoid duplicates. This is repeated until all pointers point past the end of their arrays.	
Ordered lists not in ordered list	Finds elements in a list that are not in any of a list of lists. Initialises a pointer to the start of the original list and to each of the list of lists. Compares each value pointed to against the value pointed to in the original list. If a pointed value is less than the original pointed value, its pointer is incremented. If all pointed values are greater than or equal to the original pointed value the original pointer is incremented. If no value was equal to the original value it is appended to the result	

Search Functions

The search function designs centre around removing the need to search through the whole document since this was assumed to be the largest factor impacting runtime efficiency (more so than number of sections, occurrences of particular words etc).

Search Function	Methodology	Justification
wordCount	Traverses down documentTrie for the search word. If the word can't be fully traversed it doesn't exist in the document and returns 0, otherwise it returns the length of the list of occurrences of the word stored in the terminating node.	Reduces complexity from O(document.length) to O(word.length) since it now traverses down documentTrie instead of traversing document
phraseOccurrence	Traverses down documentTrie for the first word in the phrase. If the word can't be found then the whole phrase can't ever occur and hence returns an empty list, otherwise it searches the text starting at each occurrence of the first word. textLines allows for O(1) access to the text at the position of the occurrence of the first word	Removes need to search through whole document. Can now search through beginning at places known to have the correct starting word. Could implement a KMP search or equivalent to do search starting at each occurrence of first word but since the match has to begin at the specified position, the naïve approach is just as efficient.

prefixOccurrence	Traverses down documentTrie for the node that terminates the prefix. If this node can't be found then the prefix doesn't occur and an empty list is returned. Otherwise, it performs an inorder traversal of the subtree rooted at the node that terminates the prefix. The occurrences at each node in the subtree are accumulated, and returned.	Tries naturally store prefix relationship between words and allows us to narrow down search to subtrie rooted at end of prefix. Alternatively, could perform KMP or equivalent search for prefix but wanted to remove time complexity dependence on document length. This said, this implementation would be simpler since it could call phraseOccurrence
wordsOnLine	For each word in words, the list of occurrences of the word are found from traversing the documentTrie. If any word cannot be found or has no occurrences, an empty list is returned since a valid line must contain each word. The *intersection of ordered lists* algorithm described above is used to find the lines that all words occur on and this intersection is returned	Use of trie removes time complexity dependence on document length. *Intersection of ordered lists* is justified above. Modularity in design, *wordsOnLine*, *someWordsOnLine*, *wordsNotOnLine* implemented very similarly
someWordsOnLine	For each word in words, the list of occurrences of the word are found from traversing the documentTrie. The *union of ordered lists* algorithm is run on all non-empty lists of occurrences and this union is returned.	Use of trie removes time complexity dependence on document length. *Union of ordered lists* is justified above. Modularity in design, *wordsOnLine*, *someWordsOnLine*, *wordsNotOnLine* implemented very similarly
wordsNotOnLine	The *wordsOnLine* method is used to find the lines containing all required words. The *ordered lists not in ordered list* is then used to isolate those lines which also do not contain a word from excluded words.	Use of trie removes time complexity dependence on document length. *Ordered lists not in ordered list* is justified above. Modularity in design, *wordsOnLine*, *someWordsOnLine*, *wordsNotOnLine* implemented very similarly
simpleAndSearch	Section numbers for the titles is generated using sectionIndexes. A HashSet of section numbers for each word of the sections they occur in is obtained from traversing down documentTrie. Each of these HashSets is queried for each section number. If all HashSets contain a section number then all occurrences in that section of all words is appended to the result. The occurrences in the section number are found from performing a binary search on an ArrayList of section numbers for that word which correspond to the ArrayList of occurrences.	Use of trie removes time complexity dependence on document length. Use of HashSets allow for O(1) queries. Simple logic operations performed on results of these queries
simpleOrSearch	Section numbers for the titles is generated using sectionIndexes. A HashSet of section numbers for each word of the sections they occur in is obtained from traversing down documentTrie. Each of these HashSets is queried for each section number. If a HashSet contains a section number then all occurrences in that section are appended to the result. The occurrences in the section number are found from performing a	

	binary search on an ArrayList of section numbers for that word which correspond to the ArrayList of occurrences.	
simpleNotSearch	Section numbers for the titles is generated using sectionIndexes. A HashSet of section numbers for each word of the sections they occur in is obtained from traversing down documentTrie for both the required and excluded words. Each of these HashSets is queried for each section number. If all required HashSets contain a section number and all excluded HashSets do not then all occurrences in that section of all required words is appended to the result. The occurrences in the section are found by performing a binary search on an ArrayList of section numbers for that word which correspond to its ArrayList of occurrences.	
compoundAndOrSearch	Section numbers for the titles is generated using sectionIndexes. A HashSet of section numbers for each word of the sections they occur in is obtained from traversing down documentTrie for both the *and* and *or* words. Each of these HashSets is queried for each section number. If all *and* HashSets contain a section number and at least one *or* HashSet contains it then all occurrences in that section of all *and* words and any matching *or* word is appended to the result. The occurrences in the section are found by performing a binary search on an ArrayList of section numbers for that word which correspond to its ArrayList of occurrences.	