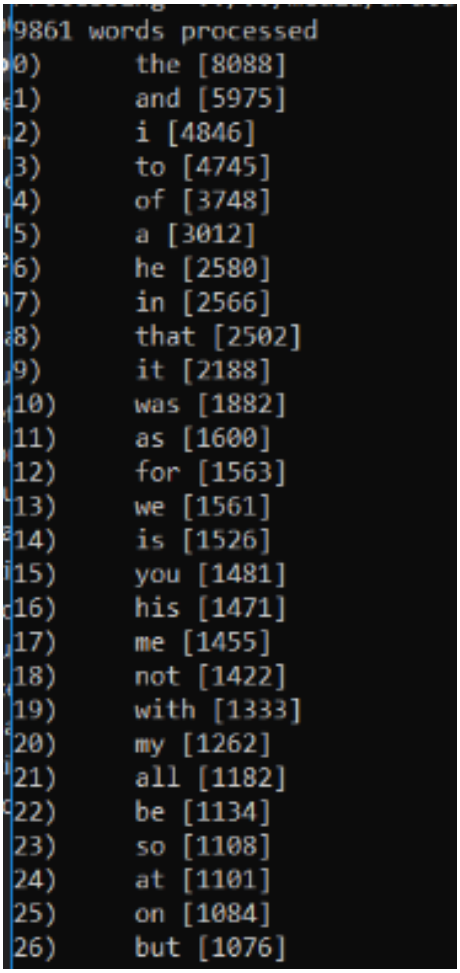
ETEC2101	Lab8: **Maps (Hash Tables)**	**100 points** (140 points possible)

1. Implement the Map class as discussed in class:
    a. (**10 points**) **General structure**[1], attributes, node (or pair?) class. A few options:
        i. You could inherit from ArrayList (use private inheritance) – a "is a" relationship
        ii. You could use a "has-a" relationship (make an ArrayList attribute). You must be very careful, though, to change the way the grow is done (it should be based on load factor, and the way we grow is very different)
        iii. [this is what I did] Just make your own *array* (not ArrayList) internally.
        iv. Whatever you choose, we need a way to identify if a spot is "used" or not.
    b. (**10 points**) **Constructor / Destructor**
    c. (**15 points**) **ostream operator**
        i. This is optional, but I wrote another method called "debug_output" which shows all the key-value pairs, including nulls. Useful for testing!
        ii. I want the ostream operator method to NOT show the null entries, but if written properly this method could just call debug_output to do that.
        iii. Desired output (python-style) for a string => int map might be:
        `{Bob:42, Sue:99, Carl:-1}`
    d. (**10 points**) **[] operator** (I want this to be the [only] means of adding as well as getting…)
        i. Make sure to properly "grow"
    e. (**10 points**) **contains / find**
        i. if you have the iterator, implement find, which should take a key and return an iterator to the found pair or end if it wasn't found
        ii. If you don't implement the iterator, implement contains which takes a key and returns true if we have a pair.
        iii. This might seem redundant considering [], but the [] will *create* the pair if it doesn't not exist (this method shouldn't)
    f. (**10 points**) **remove**
    g. **(10 points)** test code
2. Some additional features
    a. (**20 points**) iterator
    b. **(10 points)** *good* Documentation
    c. (**35 points**) An application of a data structure! Using our Map class, open a text file (I'd recommend a large-ish one from "Project Guttenberg")
        i. Break lines of text up into words. I found the isalpha, isspace, and ispunct functions helpful as well as the std::string::append method.
        ii. Then after the frequency table is built, print out the top n words (make sure the user can change this easily). You can use our sorting algorithm from lab4 to get the top words!
            1. I stack overflowed with quicksort so had to use bubblesort ☹
        iii. You can compare your values to https://www.online-utility.org/text/analyzer.jsp (scroll to the bottom to see the word-counts). My values were slightly off, but they should be close (my algorithm in step a might be slightly off). For example, I read in "Dracula" and got these frequencies (to the right is the "correct" count from the website)
        iv. (I have my results on the next page on the left, the analyzer's results on the right)

[1] You can use inheritance on this one (from ArrayList), but if you do make sure only these methods are exposed. You can also use a has-a implementation or just repeat key parts of ArrayList in this class (they are different enough, that I'd say it doesn't break the DRY rule)



Unfiltered word count:

Order	Unfiltered word count	Occurrences	Percentage
1.	the	8088	4.8668
2.	and	5975	3.5953
3.	i	4796	2.8859
4.	to	4745	2.8552
5.	of	3748	2.2553
6.	a	3007	1.8094
7.	he	2567	1.5446
8.	in	2566	1.5440
9.	that	2483	1.4941
10.	it	2170	1.3058
11.	was	1882	1.1325
12.	as	1599	0.9622
13.	for	1562	0.9399
14.	we	1552	0.9339
15.	is	1524	0.9170
16.	his	1471	0.8851
17.	me	1466	0.8821
18.	you	1465	0.8815
19.	not	1422	0.8557
20.	with	1333	0.8021
21.	my	1259	0.7576
22.	all	1179	0.7094
23.	be	1134	0.6824
24.	so	1108	0.6667
25.	at	1099	0.6613
26.	on	1084	0.6523
27.	but	1075	0.6469
28.	have	1065	0.6408
29.	her	1059	0.6372
30.	had	1039	0.6252