Introduction

The primary goal for this project was to create a web crawler that would be able to crawl though webpages and store data about the webpages in an efficient and easily indexable manner. In addition, this project is meant to be able to use the indexed data in order to perform simple web queries. This meant being able to implement a query language that could be parsed and used to find relevant web pages.

Some personal goals were to make my Index be able to store data with efficient space complexity that also allowed you to access the stored data with efficient time complexity. In addition, I wanted my web query to be robust and able to handle a wide range of cases and invalid queries.

Project Implementation

To implement this project I broke it down into 3 main parts. The two obvious parts are the web crawling portion and the query parsing portion, but another challenge that I decided to separate out was deciding how to implement the WebIndex. The WebIndex was challenging because it needed to be able to support easy data accessing for query parsing while also be able to facilitate the crawling function and be space efficient.

WebIndex

For my implementation of WebIndex I took some inspiration from my implementation of the dictionary in the Boggle project and used a HashMap that used words as the key. While this would require a lot of space to store every word, I thought it made the most sense since it would allow query parsing to be much more efficient and straight forward.

For the values in the HashMap I had to consider what the query parser would need to return results. The most obvious thing that needs to be stored for each word is the list of URLs that contain the word. One challenge that would need to be overcome was figuring out how to support phrase queries since they would require maintaining data on the order of words in the WebPage. My solution to this problem was to maintain a list of the locations where the word appears in the specific webpage. To implement this solution, I decided to make the values a nested HashMap that used the URLs as keys and a HashSet of the word indices as the value. All together my main data structure in WebIndex was a HashSet nested in a HashMap nested in another HashMap which stored the exact location of where each word was in each URL.

Additionally, my WebIndex stored a String HashSet of all the URLs. This because useful in Negation Queries as well as preventing the crawler from crawling a web page more than once. I decided to use Strings to represent the URLs in all of my data structures because there were situations where the equals() method for the URL class did not work as I wanted it to which made it not function properly when used in a HashMap or HashSet.

My WebIndex also had a lot of helper methods to facilitate crawling and queries. Here is a detailed explanation of each of the helper methods:

- **add() -** This method takes in the word, the url that the word was in, and the index location where it occurred and adds it to the HashMap.
- **addURL() –** This method takes a url and adds it to the HashSet of URLs.
- **getAllURLs() –** This method returns the HashSet of all the URLs.
- **containsURL() –** This method checks if the given URL is contained in the set of all urls.
- **getURL() –** This method took a word as a parameter and returned the HashSet value associated with the word.

Although many of these helper methods are very simple, they became useful later for reducing the amount of code and improving readability for the crawling and query functions.

Web Crawling

The actual crawling part of the program was relatively simple to implement since a large portion of the crawling and web parsing was already given in the starter code. The two main things I had to implement was 1) a way to find and parse webpages/urls that were embedded in each webpage, and 2) to collect all the relevant data and add it to the WebIndex.

To find and parse the webpages/urls I used the handleOpenElement() method in CrawlingMarkupHandler to check for "a" tags, which we were told would contain urls in the "href" attribute, and check if the url is valid. A valid url means that it has either a ".html", ".htm", or ".txt" file extension and it has not already been found before which it checks using the list of URLs in WebIndex. If the url is valid then it adds it to the master list in WebIndex and saves it in a separate list to be returned in the newUrls() method where it also gets reset.

To solve the second problem of collecting and storing the relevant data from the webpage I use the handleText() method. This method parses through the text one character at a time and checks if it is a alphanumeric character. If it is, then it's added to a variable that stores the current word. If it is not an alphanumeric character and the currently stored word is not empty then it's an indication that the word has ended and it gets added to the WebIndex using the add() method and is reset to an empty string.

Additionally, the CrawlingMarkupHandler keeps track of the current index in the document by storing an integer variable that is incremented each time a new word is added to the index and is reset each time the document ends.

Query Parsing

Similar to the other components of this project the query parsing can be broken down into two different components. These components are 1) storing the query in an easily usable data structure, and 2) parsing the query string and constructing the query data structure.

Query Data Structure

For my query data structure I decided to do something similar to what was explained in the project assignment handout which was to implement a tree. To do this I created a QueryInterface that would be implemented by classes that represented the different query types.

The QueryInterface had a QueryType enum that had each type of query and 3 abstract methods: getSatisfied(), isSatisfied(), and getType(). The purposes of these methods are detailed below:

- **getSatisfied() –** returns a HashMap of all the valid urls for that query
- **isSatisfied() –** takes in a url and returns a Boolean value of if it satisfies the query requirements
- **getType() –** returns the type of the query

Using this interface, I created a different class for each query type that implemented the three methods according to how the query functions. Those classes are explained below:

- **SimpleQuery**
  The simple query class handles queries of individual words and is by far the simplest class to implement. This class stores the word that is taken as a parameter in the constructor. For the isSatisfied() method it uses the getURL() method in WebIndex to get the HashMap of URLs. Similarly, for the getSatisfied() method it gets the list of of urls and checks if it contains the input url.
- **NotQuery**
  Since my implementation of negation queries only handles single word negations this class stores the word that is passed as a parameter in the constructor. For the isSatisfied() method it accesses the HashMap of URLs for the word stored in WebIndex and checks that the url parameter is not in the HashMap. For the getSatisfied() method it makes a clone of the HashSet retrieved from the getAllURLs() method in WebIndex and removes all urls that contain the word.

- **PhraseQuery**

  The phrase query works differently from the previous two classes and is much more complex since it involves multiple different words and also has to take the order of words into account. I decided the best way to implement this method was to turn each word in the phrase into a simple query that could be used to easily retrieve all the URLs with that had that particular word.

  For the isSatisfied() method the program stores and maintains a list of valid indices that follow the phrase in order to check that the words are all in the right order in the web page. The program starts by initializing the indices list with the list of indices associated with the url and the first word in the phrase. Then it iterates through each word and checks if the index list for that word has a value that is one more than a number stored in the current list of valid indices. If at the end of iterating through all the words the list of valid indices still has values then that means there was at least one instance where the phrase appears.

  To implement the getSatisfied() method I decided that the easiest and most simple method was to utilize the isSatisfied() method. To do this, the program uses a helper method called getContainsWords() to compile a list of all URLs that contain all the words in the phrase and passes it to the isSatisfied() method. If the method returns true, then it adds the URL to a list that is returned at the end.

  The getContainsWords() method functions by storing the set of urls that are associated with the first word and calling the retainAll() method with the set of urls for all the other words. This results in a set of URLs that only has urls contained in the sets of every word in the phrase. This method became very helpful in reducing the amount of operations needed to determine the set of all valid urls since it reduces the number of urls that the getSatisfied() method has to iterate through.
- **OrQuery**

  To implement or queries I decided to have this class store QueryInterface objects that represent the left and right query. The isSatisfied() method uses the isSatisfied() methods of both of its children and returns true if either of them return true and false otherwise. The getSatisfied() method also uses the getSatisfied() method from each of its children by storing the urls from the left query and adding all the elements in the right query. Since I decided to use a HashMap to store the urls, the program doesn’t need to perform any additional operations to prevent duplicates of each url.

- **AndQuery**
  The and query operates very similarly to the or query in the way that it stores information. The primary differences are of course that the isSatisfied() method checks that both children return true and that the getSatisfied() method uses the retainAll() method instead of putAll() method for the HashMap.

Parsing Query String

In order to actually parse the query string and build the query tree I decided to use various helper methods. The main method that I used was the buildQueryTree method which parsed through the string and returned a query object.

The general overview of how the method works is by iterating though each character in the string and performing a particular action based on what the character is. I also decided to use recursion in order to breakdown the query into subqueries that could be parsed independently. To prevent the method from iterating through parts of the query string multiple times, I chose to use a Character ArrayList that would be passed to the method and that the method would remove the character after it was parsed. By making this decision, it simplified the code since the program would always check the first item in the ArrayList and wouldn't need to keep track of the index and, it made it impossible to repeat parsing though a section of the query when recursing.

In addition to the character ArrayList, the method also keeps track of a few other pieces of information:

- q – the character ArrayList
- type – stores the QueryType whenever an "&" or "|" character is parsed, null otherwise.
- subQueries – an array of size 2 that stores the left and right side of a query.
- currentWord – the word that has been parsed so far.
- negate – stores if a "!" has been parsed for the currentWord.

This method also utilizes many helper methods that makes the program more readable and less redundant. The methods are described below:

- **addSubQuery() –** This method adds a query to the subQueries array. To do this, it checks whether type is null which indicates if the query should be added to index 0 or 1 of the array. If the index that it should be added to already has an object stored (implying that there is an implicit and) then it creates an AndQuery object with the previously stored value and the new query object.
- **addSubQuery() (again) –** I also made an overloaded version of this method that takes in a word and the negate variable instead a query object. Using these two variable it creates either a NotQuery object or a SimpleQuery object and uses that to call the other addSubQuery() method.

- **buildQuery() –** Takes in the subQueries array and the type variable to create a query object. If the subQueries array has no object in the 0th index then it returns null. If it has no object in the 1st index but has one in the second then it simply returns the query in the 0th index. Otherwise, if both indices has an object then that implies that there is a type value that it uses to create either an AndQuery or OrQuery object using the two items in the subquery array.
- **buildPhraseQuery() –** This method takes in the character ArrayList from buildQueryTree() and parses until it finds a closing '"' character. It finds all the words (alphanumeric character chunks) and adds it to an ArrayList to be passed to the PhraseQuery constructor.

Now I will detail what the buildQueryTree does for each character case:

Character	**Function**
(	Removes the first item in q, calls buildQueryTree() with q, then adds the returned value using addSubQuery() The recursed call of buildQueryTree() will remove all characters that it parses which should be everything including the closing parentheses
"	Indicates the start of a phrase query. Calls the buildPhraseQuery() method and passes the result to addSubQuery() method. The buildPhraseQuery() method will remove all character that it parses which should be everything inside and including the quotation marks
Alphanumeric character	Indicates that it is part of a word is added to the currentWord string
!	Sets the negate variable to the inverse of its value
&	Checks if the type variable already has a value. If it does then it indicates there is an error and the method returns an error state. If the type variable does not already have a value then it sets the type variable to AND and checks if currentWord has a value that needs to be added to the subqueries array
|	Checks if the type variable already has a value. If it does then it indicates there is an error and the method returns an error state. If the type variable does not already have a value then it sets the type variable to OR and checks if currentWord has a value that needs to be added to the subqueries array
)	Indicates the end of a query/subquery. Checks if currentWord has a value that needs to be added to the subqueries array then calls the buildQuery() method to construct the query object to be returned.

Anything else	This would mean there's a non-alphanumeric character that isn't a specific token and therefore it could indicate the end of a word. If currentWord is not empty, then the word is added using the addSubQuery() method.

If the method reaches the end of the character ArrayList then it also runs a similar set of operations as ")" and returns the query.

Once the query object has been made then the query() method calls and returns the getSatisfied() method from the object.

Error Handling

Since a large portion of the web crawling part of this assignment was already written for us, there wasn't very many error states that I needed to handle. The main problem that I had to consider was if a malformed url was found while parsing, in which case I decided to just ignore the url and not return it. While the program could have printed out an error message, I decided that it made more sense not to since it was likely that the crawler would encounter large numbers of urls and many of them would be malformed. Therefore, it would not be reasonable or helpful to print an error message each time.

For the webserver and queries on the other hand, there were a lot of possible error states that needed to be handled. For all invalid queries, I decided that the best course of action was to return null. Then if null was every received when a query object was expected then it would also return null. On the highest level in the WebServer class, if null was received instead of a set of pages then it prints onto the server screen that an invalid query was inputted. The following is a list of all the error states that the program checks for:

- no closing quotation mark for a phrase query
- multiple "&" or "|" in a set of parentheses
- a null or empty query
- a "!" is in front of something that isn't a word

Testing

In order to test my program I decided to use a combination of white box and black box testing. For a majority of my black box and general testing I put in various input queries and checked that the result seemed correct based on the number of urls returned. To make this testing easier I modified the webserver starter code to display the number of urls found.

Black Box Testing

One helpful tool that I used was the excel sheet posted on Ed Discussion by Annie Hu that had a lot of queries and the number of results that other people's programs outputted. When using these tests I made sure to pick a wide range of test that tested several permutations of queries such as nested queries, and each type of query.

In addition, I decided to use black box testing to test invalid queries. The invalid queries that I listed in the error handling section of this report were all tested with multiple different words and permutation and each time it was checked that the webserver showed the error message and resulted in no pages.

White Box Testing

For a majority of white box testing I decided to make my own smaller subset of html files to be parsed. By doing this I was able to make the tests more isolated and easier to verify. To make this test I created 3 html files and 1 htm file with some text and ran tests that I knew the results to. The primary goal of my tests were to ensure that each permutation of query was working properly which I measured by the number of web pages that it returned. To make sure that the tests were returning the correct number by chance I made sure to test multiple examples of each permutation and that each test had a different number of results. This helped ensure that the tests actually verified the correctness in the most efficient manner.

Another thing I made sure to include in my tests were different combinations of spaces and parentheses to make sure that the tests were robust and didn't miss any edge cases. The tests I ran are individual tests for each type of query, and multiple combinations of tests with the different queries. Each test resulted in the expected number of web pages returned.

While I could have also tested specific functions within the WebQueryEngine class, such as my buildQueryTree method, I decided that it was unnecessary and not worth the time that it would take. Since the query tree is so complex and has a lot of depth, it would be difficult to confirm the structure of the tree through automatic testing. Additionally, I felt that since the query process was so complex, that testing simply though the resulting web pages was sufficient in checking that every step through the process was correct.

Although I did not write automatic tests for checking the structure of the tree, I did run the query method through the debugger for a few manual tests and checked through the debugger that the query tree structure for those particular tests matched what I expected.

Conclusion and Overview

Problems

Throughout the process of writing my code one of the most difficult problems that I had to overcome was the sheer amount of complexity in the assignment. Unlike the previous assignments, I could not immediately begin coding with whatever ideas were in my mind and actually had to spend a few days just going over the instructions and understanding the structure of the program. Additionally, since the WebCrawler and WebServer both relied on WebIndex, it was difficult to come up with a WebIndex design that would work well with both programs.

The main problems I had with testing were with how big the test samples were and figuring out how to verify that my program was functioning correctly. Since president96 had so many URLs it is nearly impossible to know for sure what a certain query should result in. One thing that did help overcome this problem was the shared google sheets posted on Ed Discussion by Annie that allowed me to check my results with other students.

Limitation and Scope

One limitation with this program is that it takes up a lot of space when storing the WebIndex which may become more of a problem when trying to parse larger sections of the web. Additionally, the information that this program currently stores is limited only to the text and the url which means that it can't store or provide information on things like images or titles.

Additionally, the query language is pretty limited in handling some error/edge cases which means that the user must adhere to the strict rules in order to get results from the webserver.

The scope of this project is pretty broad and we can clearly see in our everyday lives of how a program like this can be used for things like Google and Safari. While my program specifically is limited to the urls contained in the file system, it can easily be expanded to include all urls on the web. This makes it an incredibly powerful tool that can be used to gather and find information about specific topics very quickly.

Time Complexity

The time complexity for WebCrawler is O(n) where n is the number of characters in the files since I iterate through each character to build the WebIndex. Most of the other parts of WebCrawler were given in the starter code so the complexity is not known to me since I don't know how it's written.

For the time complexity of building the query tree it takes O(n) time where n is the number of characters in the query since the main operation being run is iterating through the character ArrayList.

To retrieve a list of urls from the query, the time complexity depends heavily on what type of query is being run. Here are all the time complexities:

QueryType	GetSatisfied	isSatisfied
Simple	O(n) – to convert each url to a page object	O(1) – accessing element in hashmap
Not	O(n) – to remove items from the list and convert url to page object	O(1) – accessing element in hashmap
Phrase	O(*number_of_words* x n^2) – iterating through each url and running isSatisfied	O(*number_of_words* x n) – iterating through each list for each word
Or	O(n) – accessing the getSatisfied method from each child	O(1) – accessing each element
And	O(n) - accessing the getSatisfied method from each child	O(1) – accessing each element

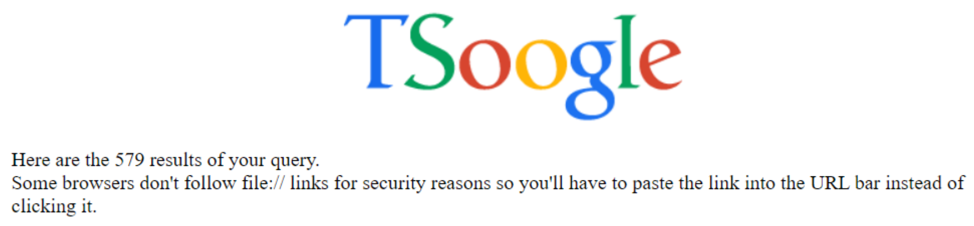
Karma

As a result of my implementation method for query parsing, the requirements for a valid query were much more relaxed for my program than detailed in the project assignment.

- Since my negation check toggles the value of the negate variable, it is able to support multiple concurrent "!".
- Since my program recurses every time it encounters a ")" and returns every time it encounters a ")" it is able to function even when there are multiple redundant sets of parenthesis.
- If there are open brackets and no close brackets then the query will still be able to be built and it assumes that the open brackets get closed at the end of the query
- Because my program updates the subQueries list as it parses the query string, it is able update the subQueries inside nested queries which means that it can support implicit ands at any point in the query including implicit ands between two queries.

Interesting Results:

I tried to find the word with the most urls and ended up finding that the word "the" is in 579 webpages in president96.



Here are the 579 results of your query.
Some browsers don't follow file:// links for security reasons so you'll have to paste the link into the URL bar instead of clicking it.