Overview

In prog7, we were required to create a web crawler that would start at one HTML file and find other HTML files from the original file, and repeat this process until all files have been found branching from the original file. In the files, we were required to store each individual word (where a word is any alphanumeric text separated by whitespace) so that we could create a functioning search engine that would return the HTML(s) in which a user inputted query was found in.

With this in mind, my goals for this project were to become familiar with or better with the shunting yard algorithm, HTML tags (specifically how to find text and URLs in an HTML file), nested data structures, JUnit testing, set theory, and string manipulation.

This was an incredibly difficult project for me and I underestimated the difficulty of evaluating the operations of a query. I also pulled two and a half all nighters consecutively (with naps) to finish this project 🥳 and spent pretty much all my time in those four days working on web crawler. Every second I wasn't working was thinking about web crawler. I have become a web crawler. I think I've eaten a collective three meals and lots of hi-chew and chips across these days and have either been in PCL or my dorm. This was all a result of my poor time management, an overestimation of my abilities, and an underestimation of how difficult the search operations would be to implement. This project has felt like a marathon but if I had to sprint it.
So, lesson learned, do not procrastinate, especially on web crawler.

Solution Design

The first thing I needed to figure out with this project was how each class interacted with each other. From office hours, I learned that `CrawlingMarkUpHandler` was the class that would go through each document and handle elements and text which would be useful for parsing URLs and words. I learned that `WebIndex` is my data structure that should store the parsed URLs and words in a meaningful way for me to access and support search/traversal operations of the data structure. Lastly, I learned that `WebQueryEngine` is the class that processes user inputted queries and should somehow search the queries in `WebIndex` to return the URLs of which the query is found in.

After understanding how each class interacted with each other, my approach/steps for this project were roughly this:

1. Start on `CrawlingMarkUpHandler` to parse the files for URLs and text
2. Create my index data structure in `WebIndex` to store words, the URLs the words are found in, and the location of the words in the URLs indexed from 0
3. Make sure that my `CrawlingMarkUpHandler` is updating my index

4. In `WebQueryEngine`, parse the user inputted query by tokenizing then putting it in reverse polish notation using the shunting yard algorithm
5. In `WebIndex`, create search operations for & and | to evaluate the shunting yard
6. Create search operations for ! (nots)
7. Create search operations for phrases
8. Tokenize implicit ands to allow for the & operation
9. Know when to use the operations to evaluate the shunting yard-ed query and what to do with the results (did this throughout the process of creating the search operations as well)
10. Lots of debugging throughout this search operation creation process

Steps 4, 6, 7, 8, and 9 were the most difficult, painful, frustrating, and horrible to implement.

Steps 1 and 3: `CrawlingMarkUpHandler` for Parsing Files and Storing in `WebIndex`

I combined explaining steps 1 and 3 because they go with each other, step 1 is parsing and step 3 is storing from `CrawlingMarkUpHandler`.

In this initial step, I used the handle___ functions to tell the attoparser what to do when a document is opened, when an element is opened, when an element is closed, and when text is found. I also created my own function for keeping track of the page that the parser is currently on which is needed for adding the page to the index.

When parsing files, I am looking for unvisited URLs and text in each document. For this process, I have to make sure attoparser isn't looking in "script" and "style" tags because they contain irrelevant information for a search engine purpose, but attoparser still picks up on them.

To make sure that attoparser isn't looking in "script" and "style" tags, I keep a counter of "script" and "style" encounters in an integer called `scriptStyleTags`. In `handleOpenElement`, if the tag is "script" or "style", increment the `scriptStyleTags` int. Then, in `handleCloseElement`, if the tag is "script" or "style", decrement `scriptStyleTags`. That way, when scriptStyleTags is 0, it means that attoparser is NOT looking in a script or style tag element and instead, the tag could contain important information such as a URL or text. It was important to use a counter instead of a boolean because there could be nested "script" and/or "style" tags.

To find URLs, in `handleOpenElement`, I look for "a" tags with the attribute "href". I then add the URL to a HashSet that tracks all of the visited URLs to make sure there are no cycles of URLs and my WebIndex.

To find text, in `handleText`, I append all the text to a StringBuilder and split on all whitespace and non alphanumeric characters. Splitting on non alphanumeric characters ensures that text that appears like “Clinton/Gore” becomes “Clinton” and “Gore”. However, it also meant that any contractions, for example “Clinton’s”, became “Clinton” and “s”. I found this to not pose an issue however because valid queries were only alphanumeric, so searching contractions weren’t an option and in fact, storing “Clinton” would be more valuable in that case. However, this also meant if “s” was a word found on a page, the number of “s”s could be overcounted. Also, much later in the implementation process, I realized that my program was storing “ ” text as a word which also didn’t pose an issue, but would be technically incorrect because “ ” doesn’t appear as text on an HTML. After splitting, I would store the text in my `WebIndex` along with it’s page and location. When keep tracking of a word’s location on a page, I just kept an arbitrary counter called `wordLocation` as an integer that resets to 0 upon every new document in `handleDocumentOpen`. After adding a word, `wordLocation` increments. This allows me to keep track of each word’s location on the page relative to each other which is useful for phrase searching. For example, if the first text that appears on a page is “this is a new page”, then “this” would be given a `wordLocation` number of 0, “is” would be 1, “a” would be 2, etc.

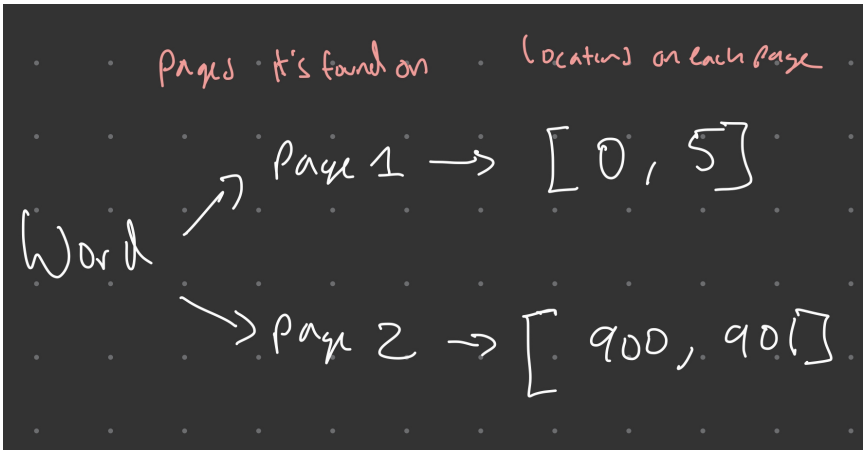
This process would continue for the URLs found on the first page, then for the URLs found on those pages, etc.

Conclusion: All HTML files have been successfully parsed for individual words and URLs! All words, URLs, and word locations have also been stored in the index.

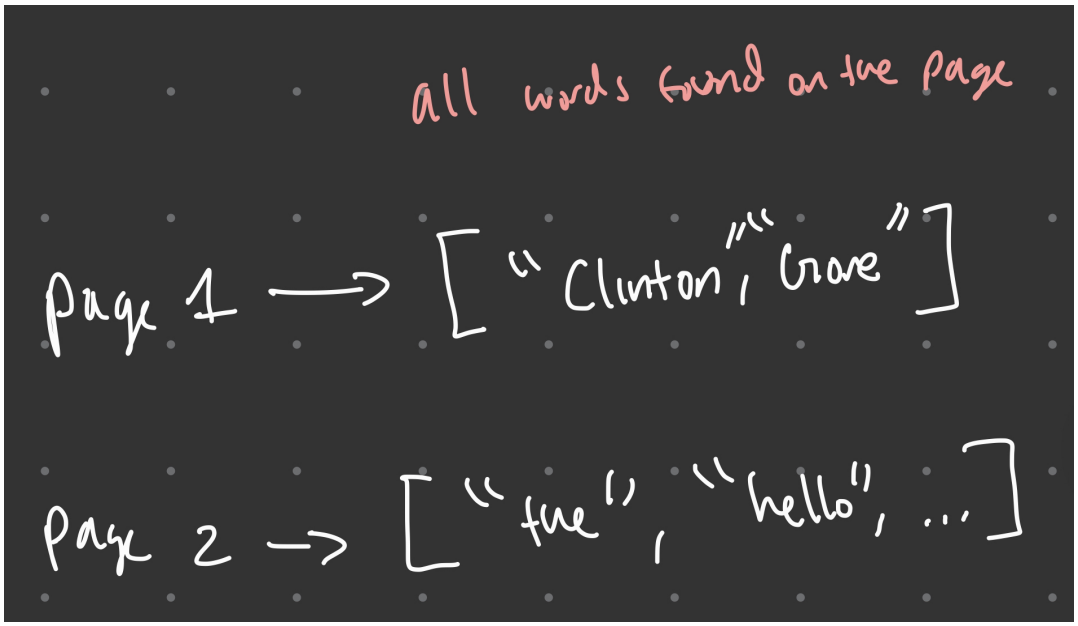
Time Complexity: The time complexity for all methods except handleText are constant because there are no loops or traversing, just constant operations. The time complexity for handleText is O(characters found on each page + number of words on each page) = O(n) because I use a for loop for each character that attoparser comes across to append to a StringBuilder and then for each word, after splitting the StringBuilder on whitespace, is added to my index. So, there are two for loops involved which is linear time.

Step 2: Creating my index data structure in `WebIndex`

After parsing in `CrawlingMarkUpHandler`, I needed to consider how to store the data parsed in which it would be useful for searching for a query. I decided that storing “pages and word locations per page” per word would be the data structure that makes the most sense for query searching because it would allow me to obtain the pages and the word’s location(s) on each page when looking a word up. Later on, for phrase searching, I realized that implementing the reverse would also be useful (storing the words per page).



A representation of how I want to store the data per word. This data is stored for every word found on every page.



The second data structure I described that is useful for phrase searching because it allows me to lookup a page and find all the words.

```
69 usages
private HashMap<String, HashMap<Page, ArrayList<Integer>>> words = new HashMap<>();
4 usages
private HashMap<Page, ArrayList<String>> allPages = new HashMap<>();
```

Lol 69 usages nice. So, my first data structure is a HashMap where the key is a word and the value is a HashMap of the pages per word and the value is the word's locations per page. My second data structure is a HashMap where the key is a page and the value is all of the words on

that page. I used ArrayLists for word locations per page and words per page because it allows me to add to them easily.

To allow `CrawlingMarkUpHandler` to add to these data structures, I implemented an `addWord` and `addPage` function that would update “words”, the name of the first data structure I described, and “allPages”, the name of the second data structure I described.

Conclusion: I now have an effective way for looking up a word and returning it’s associated pages and word locations associated with each page. I also now have an effective way for looking up a page and returning each word on the page which is useful for phrase searching.

Time Complexity: `addWord` and `addPage` are both constant, there’s no traversing of data.

Steps 4 and 8: Shunting Yard Algorithm (tokenizing and postfix-ing queries) in `WebQueryEngine`

When deciding on how to parse queries, there were pretty much two options: recursive descent or shunting yard. Although the packet talks about recursive descent, I used shunting yard because Ujjaini said it was easier. They are both **O(n)** where n is the length of the query since either algorithm should reach every token in the query, so because there is no time complexity benefit, I decided to implement the algorithm that is easier (at least according to Ujjaini 😅).

After deciding my approach for parsing queries, Shunting Yard required me to first tokenize the query. This meant a query like `((clinton & gore) | “phrase lol”)` would become the string array `[(, (, clinton, &, gore, ), |, “phrase lol”]`.

First, I split on all whitespace. This allowed an inputted query like `(clinton & gore)` to become the string array `(clinton, &, gore)`.

Tokenizing and figuring out implicit ands

Before going any further, I had to consider how I would process implicit ands. My approach with implicit ands was to insert an “&” symbol when there were two operands next to each other. For example, the query `clinton gore` would become `clinton & gore`, which are equivalent expressions. I knew that two operands next to each other would mean there is an implicit and, because if there was an operator, then those operands would be evaluated using that operator. Sounds simple but proved surprisingly difficult.

It was tricky to do this because I was trying to add to the ArrayList of the query split on whitespace whilst iterating through the ArrayList, which would create an infinite loop. So, I

instead found the indices where an “&” would need to be inserted, and then inserted them after. However, another problem I encountered with that approach was that queries that involved phrases like `“this is a phrase”` would then have an & placed between `this` and `is`, `is` and `a`, and `a` and `phrase`. So, I had to create a boolean that would track whether we’ve entered a phrase in the query by checking whether the first and second characters are `(“` or the first character is `“`. Then, I know we’ve entered a phrase. Once we’ve exited a phrase, the second to last character and the last characters are `“)` or the last character is `“`, then we continue the implicit and indicies checking process. But then, I encountered yet another problem when trying to insert the &s because I was modifying the indices as I was inserting. So, I made an integer variable, initialized to 0, that would keep track of how many times an “&” has been inserted where an implicit and was. That way, I would update the index of which the “&” would need to be added to by doing the original index the “&” was supposed to be inserted + the number of times an “&” has already been added. I would then increment this counter by 1 after every time an “&” has been added. After the implicit ands have all been inserted as “&”s, I split on all parenthesis but preserve the parenthesis using the regex `"((?<=\\))|(?=\\))|((?<=\\()|(?=\\()))".`

So, if my original query was something like `!word1 word2 (clinton & gore) “phrase lol”`, it now is a string array that looks like `[!word1, &, word2, &, (, clinton, &, gore, ), &, “phrase, lol”].`

Post-fixing

Next, I have to put this tokenized query into postfix notation otherwise known as reverse polish notation otherwise known as shunting yard.

To do so, I would iterate through each string of the tokenized query and check to see whether it was a parenthesis `(` or `)`, `&`, `|`, a phrase, or just a normal alphanumeric token. The output would be represented in a queue and the operators would be represented in a stack until it’s pushed into the output queue.

When an operand is found, it is pushed into the output. When the algorithm comes across an operator, it will be pushed into the operators stack. But, before pushing an operator, the algorithm first evaluates whether it needs to pop a previous operator from the stack into the output queue. To do so, it checks which operator has higher precedence. Higher precedence is marked by:

- top of stack is “&” and you are trying to push a “&”
- top of stack is “&” and you are trying to push a “|”

If the top of the stack has a higher precedence than the operator you are trying to push into the stack, then the top of the stack gets popped and placed into the output queue, then the lower precedence operator gets pushed into the stack.

If you come across a left parenthesis, then that automatically gets pushed into the operator stack no matter what because a left parenthesis always has higher precedence than any other operator. Once a right parenthesis is encountered, all operators are popped into the output queue until it's corresponding left parenthesis is found. Then, you perform one last pop of the stack to get rid of the left parenthesis in the stack. Phrases are also pushed into the output queue, but I've added some conditions to make sure the phrase is added as a whole and without parenthesis using a StringBuilder.

This entire process puts the tokenized query into post-fix notation. Post-fix notation is useful because it allows us to evaluate an expression with the correct order of operations with only the operators and operands (no need for parentheses) in a left to right scanning fashion.

To outline how this process works, let's say we're trying to post-fix the tokenized query: `[(, !word1, &, (, "dole, kemp", ),)]`.

1) First token "(": gets pushed into operators stack
    - operators stack now: (
    - output queue:
2) !word1: gets pushed into output queue
    - operators stack: (
    - output queue now: !word1
3) &: checks whether stack's top has a higher precedence than this operator, it doesn't, so & is pushed to stack
    - operators stack: (, &
    - output queue: !word1
4) "(": pushed into operators stack
    - operators stack: (, &, (
    - output queue: !word1
5) "dole: algorithm sees that this is the beginning of a phrase, so then finds end of phrase to construct the whole phrase. It sees that the next term is kemp", marking the end of the phrase because the last character is a quotation. So, the StringBuilder constructs the new phrase to add to be `dole kemp`.
6) dole kemp: pushed to output queue
    - operators stack: (, &, (
    - output queue: !word1, dole kemp
7) ")": found a right parenthesis, so pop from operators until left parenthesis is found and pop left parenthesis
    - operators stack: (, &
    - output queue: !word1, dole kemp
8) ")": repeat same process as above, this time the & gets popped in the process of finding a left parenthesis
    - operators stack:
    - output queue: !word1, dole kemp, &

So, we've successfully post-fixed `(, !word1, &, (, "dole, kemp", ),)` into `!word1, dole kemp, &.`

Conclusion: We've now successfully tokenized a query and put the query into post-fix notation, shunting yard, for it to be evaluated.

Time Complexity: The tokenizing process is O(n). With consideration of implicit ands, 6 for loops are used, none nested, so it would be roughly O(6n) where n is the number of elements in the query. Not very optimal, I know, in fact, I now realize that I could've done without one of the for loops in the implicit and tokenizing process. Putting the query into post-fix notation is just O(1n), where n is the number of tokens in the query, because the program just evaluates each element in the tokenized query left to right and decides what to do with it.

Steps 5 and 9: Creating and using the & and | search operations in `WebIndex` and figuring out when to use them in `WebQueryEngine`

Before creating the functions for & and | operations, I first had to know when to use them.

Knowing when to use search operations

So, I created a function called evaluateOutput that would go through the query we just tokenized and put into postfix notation to tell us when to perform a search operation and what to do with it.

I accomplished this by having a stack represent operands and an `ArrayList<String>` to represent the post-fixed output queue from the previous step. For every token in the output queue, if the token is an operand, push it into the operands stack. Otherwise, it is either an "&" or an "|" and we evaluate the operands previously pushed in the stack with the operator that we came across.

If it is an "&", then perform an & search. If it is an |, then perform an | search. The type of & or | search and how to perform it is dependent on these factors:

- Is there only a single operand in the stack?
    - Means that we've come across some type of form of query that looks like `"word1, word2, & or |, word3, & or |"`, where `word3` would be the single operand
    - If so, is the single operand a phrase? (indicates a phrase search)
    - Does it have a ! in front of the operand? (indicates a not search)

        - The resultant evaluation should be the previous evaluation & or | the single operand
- Are there no more operands in the stack?
    - Means that all sub expressions in the query have been evaluated and we have to evaluate the sub expressions together until the final expression is evaluated
    - Take the two previous results generated and evaluate them with either & or | search
- Are there two or more operands in the stack?
    - Will always be the case for first evaluation taking place in a query
    - Means we've come across a query that looks like "`word1, word2, & or |`"
    - Are any of the operands phrases?
    - Do any of the operands have a ! in front of it?
    - Take the two words and perform the appropriate search operation

Now that I've given an overview of what to consider before performing a search operation, I will describe how the & and | search operations work.

& search and | search

When the operator "&" is found, it means we have to perform an & search. An & search, called `andSearch` or `andSearchHH` (an andSearch between two HashSets) in `WebIndex`, would find the intersection between the two sets of pages found from the inputted "word word", "set word", or "set set" combination. I realized I could've been much smarter with my implementation using method overloading, that way I could've had an `andSearch` for "word word", an `andSearch` for "word set", and an `andSearch` for "set set" all with the same name so I wouldn't need to differentiate between the inputted parameter types and the program could just do that for me. Instead however, I created an `andSearch` that can evaluate the intersection of pages between either "word word" or "set word" and an `andSearch` that can evaluate "set set".

In `andSearch` (not `andSearchHH`), prior to finding the intersection, I need to make sure that the index I created in step 2 "words" contains the word(s) that I'm looking up. To do so, I check whether it contains the word(s) as key(s) in the HashMap and if not, return an empty HashSet of pages.

Otherwise, we are ready to find the intersection of the pages of the word word or set word.

First, I check to see which page set is smaller to optimize how many elements we need to traverse. Then, for every page in the smaller page set, if the second page set contains the page, then add it to a HashSet of pages. Then, return the HashSet of pages.

`andSearchHH` does the exact same thing without the need to check for whether the word(s) exist in the index because we are comparing two sets as input.

When the “|” operator is found, it means we mave to perform an | search. An | search, called `orSearch` or `orSearchHH` takes the same steps as andSearch and andSearchHH except instead of finding the intersection, we are just finding the union of both sets. To find the union, we just take every page from both sets of pages and add it to a HashSet of pages and return that HashSet.

Conclusion: We know when to use each search operation, what to do with the results of the search operation, and now have functional searches for the “&” and “|” operators (not ! & and ! | searches yet though).

Time Complexity: Traversing each token in a query to find what to do with each token is O(n) where n is the number of tokens. `andSearch, andSearchHH, orSearch, and orSearchHH` are all O(n). `andSearch` and `andSearchHH` are at most O(1n) where n is the number of pages of a set because there are only at most one for loop traversing the set with the least number of pages. `orSearch` and `orSearchHH` are O(n1 + n2) where n1 is the number of pages associated with one word and n2 is the number of pages associated with the second word. The time complexity of this entire process would be O(n) because for every extra token, you could be potentially performing one more linear time search operation. The O(n) in traversing each token is different than the O(n) in the search operations because what n represents in either one is different.

Steps 6 and 9. Creating ! search and phrase search operations in `WebIndex`

When either operator “&” or “|” is found in the output list and an “!” is found at the beginning of the operand token for a single operand or an “!” is found at the beginning of two operand tokens, then we know to perform an `andNotSearch` or an `orNotSearch`.

The possible type inputs for these two search operations are either “word word” or “set word” because you can’t have ! of a set, only !word. So, “word word” is for if either word has a ! in front of it and “set word” is for finding a previous evaluation’s intersection or union with a !word.

To find the !word’s pages, I find the set of all pages in my index and remove the pages that the word is found in.

So, for example, to find `!word1 & !word2`, using `andNotSearch`, I find the pages in which word1 is not in and all the pages of which word2 is not in, and find their intersection. Likewise, for an input like `!word1 & word2`, I find the pages in which word1 is not in and all the pages that word2 is in, and find their intersection.

Prior to searching, I have to do a similar checking of whether the word exists in the index. There are 4 cases to account for if a word doesn't exist:

- "!woefjwefewf & !jfoweifjwefw" (both don't exist in the index) would return the entire index
- "!fwefwefewf & !the" would return !the
- "!weojfweof & the" would return all the pages for "the"
- "!fjweofwfw & `some set of pages`" would return the `some set of pages`

This is a similar process for the `orNotSearch` except instead of finding the intersection, I find the union of `!word1 | !word2` or `!word1 | word2` or `!word1 | a set of pages`. To find the union of a query like `!word1 | !word2`, I find all the pages of the index and remove the pages that word1 is found in. Then, I find all the pages of the index and remove the pages that word2 is found in. Then, I add each page of those resultant sets into a HashSet of pages and return that HashSet. A similar process takes place for `!word1 | word2` and `!word1 | a set of pages`.

For phrase searching, I realized that I needed the additional data structure of being able to access all the words associated with a page, so `allPages` from step 2 was born.

My approach when it came to phrase searching was first to find the intersection of the pages that each word of the phrase has using `phraseSearch`. Then, still in `phraseSearch`, using `hasPhrase`, I would remove the remove the pages from the intersection of pages that don't contain the phrase.

To find the pages that contain the phrase, `hasPhrase` uses the following data: the list of words associated with the page it's checking, the list of locations per word in the phrase on that page, and the tokenized phrase.

If the location (index) of the word + the size of the phrase exceeds the size of the list of words, then the phrase obviously can't be on that page because it would exceed the number of words on the page. Once the word location has been verified to be a possible beginning of the phrase, the next word of the phrase is checked to see if it is contained in the page at the index of the previous word + 1. This process continues for all words of the phrase until all words have been exhausted and if so, then the phrase exists on that page.

`phraseSearch` is used whenever a phrase is detected in the post fix query (the token contains a space) to find the HashSet of pages associated with that phrase. Then, the appropriate operation is performed depending on the operator encountered. For example, the post fix query `"this is phrase1", "this is phrase2", &` would first run `phraseSearch` on phrase1, then run `phraseSearch` on phrase2, then run `andSearchHH` on the two HashSets generated from phrase1 and phrase2.

Conclusion: We can now search for phrases and !word for & and | operations! Meaning, we've now successfully support all possible operations and the web crawler is done!

Time Complexity: `andNotSearch` and `orNotSearch` are both O(n) for similar reasons as to `andSearch` and `orSearch` explained in the previous section. Both of the functions are traversing each page associated with a word. `phraseSearch` is O(n) where n is the number of words in the phrase. `hasPhrase` is O(n) because it traverses through each the locations of the word found on that page and each word in the phrase. `hasPhrase` is called n times where n is the number of intersected pages of that phrase's words.

Testing

I black box and white box tested the markup handler, the web index, and the query engine.

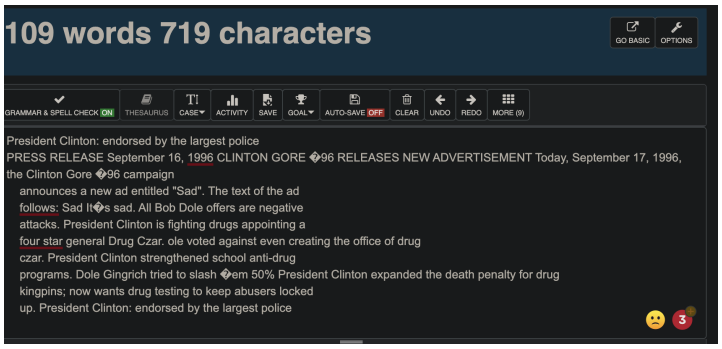
Black Box Testing

For black box testing, I would find a random URL and view it's source code to find text. Then, I would construct a query that I know would make that specific URL show up on TSoogle. For example, the URL www.livingroomcandidate.org/websites/cg96/record36.html contains the words `"Crime Bill adds 463 police officers to"`, so I query for that phrase and ensure that the URL would be returned and it is indeed returned. I repeat this process for more files and create more complex queries like for example, in www.livingroomcandidate.org/websites/cg96/dole821.html, the phrase "medicare and medicaid" is found and the word "dole" is found but not the word "kemp", so I try the query `(dole & !kemp) "medicare and medicaid"` and ensure www.livingroomcandidate.org/websites/cg96/dole821.html is returned. I would also modify the contents of a URL or create my own HTMLs to test even further complex queries with nested phrases, & or | operations, multiple implicit ands, and not operations.

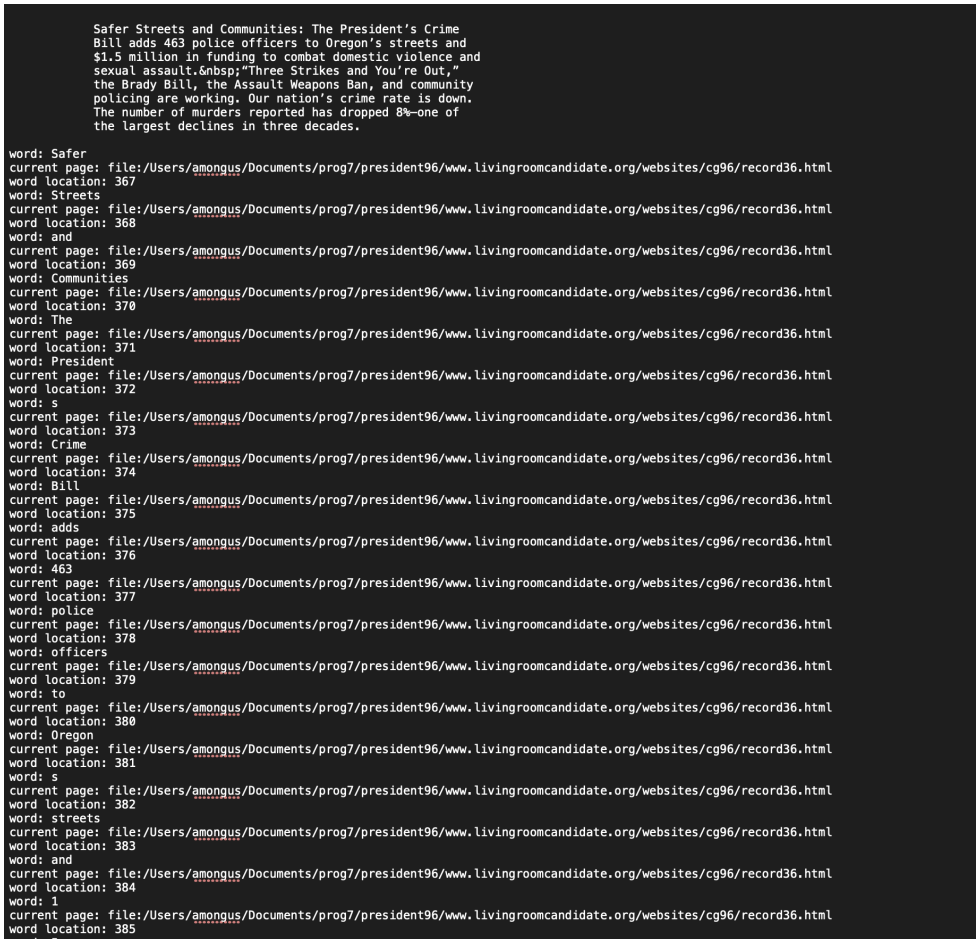
Furthermore, the Turing Scholars created a spreadsheet to keep track of the number of results that a query would return in their web crawlers. So, I tested to see if my query would either return the exact same number, or a number at least close but not the same (due to differing design decisions when handling markup or potential logical bugs ☹). For example, most of my peers received two results for the query `president available "dole kemp" !computer (actual | are)` and I also received two results. My peers received two results for `("internet is a" | danger) the (!hello | !hi)` and I also received two results. My peers received 79 results for the query `"dole kemp"` and I also received 79 results. So, I deemed my query process sufficient at least compared to my peers.

I also black box tested my `CrawlingMarkUpHandler` by starting on an HTML file of my choosing and printing out the text that it was handling, each word that was being indexed, it's word location, and the page it's found on to the console. I would then log the console by having

IntelliJ save all the print statements to a text file. I then used something like wordcounter.net to verify the word location by manually pasting the text from the HTML file into the word counter.

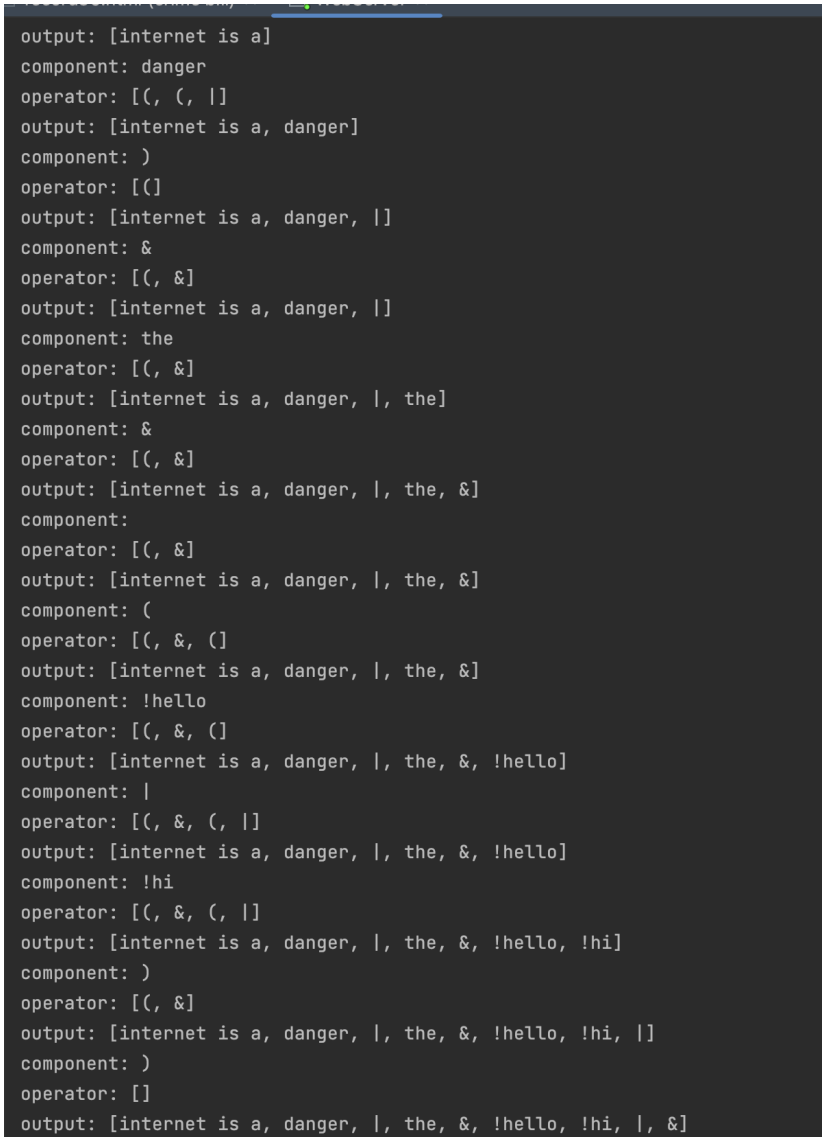


I would also verify that most if not all of the text are present on the HTML file as it says in my console and vice versa. I didn't do this for every HTML file obviously but I tried it for a few and my markup handler was consistent so I deemed it functional.



Example of console log

I was also able to black box test my query engine by printing out the tokenization process of a query and making sure that it's in the correct post fix notation. So, for example, the query `("internet is a" | danger) the (!hello | !hi)` would go through this post fix process:



This is the correct post fix notation for the query, I tried this on a bunch of other queries that involved more complex formatting (nested phrases, nested &s or |s, multiple implicit ands, nested !words, consecutive phrases, consecutive words, consecutive nots, a phrase with a word after, single operands, not words with a phrase after, and other combinations), and deemed it sufficient.

I would also make sure that a query like `(a | !a)` would return all the pages in my index and it did.

I was also able to black box test my search operations by trying out different combinations of queries, seeing if my program would reach the right search operation function using the debugger, and then seeing if the appropriate URLs returned contain the query (although this doesn't guarantee that every URL with the query was returned (but I somewhat verified this already in the first query testing process) but I can at least check all the URLs returned for the query). For example, a simple query like `"dole kemp"` should go to my phrase search automatically since there are no operators. I would then step into `phraseSearch`, make sure it's finding all the intersection pages, then going to `hasPhrase`, checking each word in the phrase, and making sure each word comes after the subsequent word in the phrase. Another example is a query like `!fwefwfewgwetthisisgibberish & the` which should go to `andNotSearch` in my code, activate `searchOneWord` for the word the, and `searchOneWord` should return all the pages present for the (which I verify by stepping into my code). I repeat this process for a variety of queries and make sure that I'm handling the possible **edge cases** in my functions.

The **edge cases/possible combinations** include: (the spam fjweojewf indicates a word not present in the index)

- `!fjweojewf & !jfowethwer` should return entire index
- `!fjweojewf & !the` should return !the
- `!fjweojewf & the` should return the
- `!fjweojewf | !jofewfw` should return the entire index
- `!fjweojewf | !the` should return the entire index
- `!fjweojewf | the` should return the entire index
- `"fjweojewf"` should return an empty set
- `"fjweojewf and the"` should return an empty set
- `"and fjweojewf the"` should return an empty set
- `"and the fjweojewf"` should return an empty set
- `fjweojewf & fwefewoiw` should return an empty set
- `fjweojewf & the` should return an empty set
- `fjweojewf | fjwefwefew` should return an empty set
- `fjweojewf | the` should return the
- `a | !a` should return the entire index
- `a & !a` should return an empty set
- `the & hello` = the & hello
- "`phrase1 phrase2phrase3"` = "phrase1 phrase2phrase3"

All the above cases are also repeated for their mirrored query, for example, the query `!fjweojewf & the` is also tested as `the & !fjweojewf`. I make sure that both queries result in the same pages returned. In this project, invalid queries weren't possible, so I didn't test how my program would handle invalid queries (I know that mismatched parentheses would result in an error however).

Normal cases are ones where the word(s) are present in the index.

White Box Testing

I used three testing classes called `testMarkUpHandler`, `testWebIndex`, and `testQueryEngine` to unit test each individual function when given edge queries, normal queries, and just verifying that functions work by giving an input and expecting an appropriate output.

In `testMarkUpHandler`, I have methods for testing the increment and decrement of `scriptStyleTags`, testing that when an <a href> is found, a URL would be added, a URL added always ends in .html or .htm, and that there are no cycles of URLs. For this process, I created my own testing suite of three URLs where one URL was the index and the other two branched from the index. They both contained easily identifiable text for me and tested edge cases like white space and if it didn't add non alphanumeric characters. They also contained script and style tags and links to each other (to test looping). I made sure that all text appeared in my test WebIndex and that it would split something like "word1/word2" as "word1" and "word2".

In `testWebIndex`, I would verify that the three pages of my testing suite were found in my first and second data structure called `words` and `allPages`. I would test `addWord` by seeing if repeated added words and URLs would just update the location, repeated added words but new URL would add the new URL and whatever location the word is at, and if it was a new word it would add to words as a new key. Likewise for `addPage`, I would do the same except if was a repeated page, the list of words would update, and if it was a new page, then the first word would be the first word of the new list for that page. I was also able to test each search function to verify it would handle the query edge cases as outlined in the black box testing section. I would also test "word word" "set word" and "set set" combinations by making arbitrary sets of pages and seeing how my program would find the intersection or union of it. For phraseSearch, I would give it an arbitrary phrase found in my test suite and an arbitrary phrase not found in my test suite. I would also individually test hasPhrase by giving it arbitrary word lists, word location lists, and phrases. Lastly, `searchOneWord` and `searchNotOneWord` were tested to see if they would find the appropriate page set of the word or the not word.

In `testQueryEngine`, I gave it a variety of queries with nested parentheses, operators, nots, and multiple implicit ands to verify the tokenization and shunting yard process. To test `evaluateOutput`, I would test empty output sizes, output sizes of size 1, and other arbitrary post fixed queries with a combination of &s, |s, nots, implicit ands, and parentheses.

Code Quality/Scope/Problems Encountered

Code Quality

I must admit, code quality was not the best with this project. If I allocated more time for myself for this project, I could've definitely cleaned up my code tremendously, however, there's lots of blocks of repeated code that could easily be replaced by helper functions. My variable naming scheme was consistently camelCase, however, sometimes the variable names would get really long. Also, there are lots of if statements that make it hard to read my code and potentially some unnecessary if statements. I also realized too late that I could use method overloading for the different parameters in `andSearch` and `orSearch` but I was just trying to finish the project at this point so I didn't bother. Instead, I used a more "janky" and unfortunately less type oriented coding method of using Object as a parameter and checking if the parameter is an instance of a set or string. So overall, not the best code quality.

Scope

I am satisfied with the scope of my project. I believe it should handle all possible valid queries and return all URLs associated with the query.

There were quite a few **design decisions** made in this project. One major one was my decision of splitting text on non alphanumeric characters as I described earlier. So, strings like "word1/word2" would instead be added as individual words into my index of "word1" and "word2". However, this did pose the problem of contractions being split too, but I decided that was fine because no queries would involved contractions and it might even be better because "clinton's" could now be found as "clinton" in my index. However, it did mean that if "s" was a word on a page, it could potentially be overcounted. Another design decision I made was that a query like "! word1" was not the same as "!word1". Another decision I made was that phrases with excessive whitespace between words would be treated as one space between words. I also treated whitespace between parentheses and words as the same as no whitespace.

Problems Encountered

Plentiful of problems were encountered throughout this project which I highlight in the solution design steps section of this report. However, I will go over some major problems I encountered.

Firstly, I didn't account for the fact that a query could be finding the intersection or union between two sets, so I didn't know how to linearly scan my post fixed query appropriately when that would be the case because there would be more operators than operands. I only accounted for `[word1, word2, &, word3, |]` but not something like `[clinton, gore, &, platform, vote, |, &]` which is the query "((clinton & gore) & (platform | vote))". As you can see, clinton & gore would be evaluated first then platform | vote, then you find the

intersection between those two sets. I solved this problem by adding to an ArrayList of HashSet of pages everytime an evaluation was made and then if there were no operands but still operands, then you find either the intersection or union between the last two sets of the results list.

Secondly, I was incredibly stumped on how to implement `phraseSearch`. However, I realized I needed to add a second data structure to `WebIndex` that could return all the words associated with a page. But then, my `phraseSearch` was bugged and it was because I implemented `allPages` wrongly (didn't allow for repeats of words on a page). Once that was fixed, phraseSearch was running smoothly.

Lots of empty stack exceptions, cases I realized I didn't account for, and debugging throughout this entire process.

Social Impact

Web crawlers and search algorithms can greatly impact people's perception of the world around them by determining which information is readily available and easily accessible. When people search for information online, the results that are returned to them are often determined by the algorithms used by search engines like Google. These algorithms use a variety of factors, such as the relevance of the information to the user's search query and the popularity of the website, to rank the results and determine which ones are shown first.

As a result, people may be more likely to encounter information that is highly ranked by the search algorithm, even if it may not necessarily be the most accurate or factual. This can create a bias towards certain viewpoints or sources of information, and make it more difficult for people to access information that challenges their preconceived beliefs or ideas.

Additionally, the use of web crawlers by search engines can also impact the visibility of certain websites or information online. Web crawlers are automated programs that search engines use to discover and index new content on the internet. If a website is not easily accessible to web crawlers, it may not be included in search engine results, making it less likely for people to find it. This can limit the diversity of information that is available to people and potentially reinforce existing biases or beliefs.

Overall, the use of web crawlers and search algorithms can significantly influence people's perception of the world around them and what they believe to be factual. It is important for individuals to be aware of these biases and make an effort to seek out diverse sources of information in order to form a well-rounded understanding of the world.

Also, people tend to gravitate towards sources that justify their point of view already.

https://www.frontiersin.org/articles/10.3389/fpsyg.2021.771948/full#:~:text=In%20cognitive%20psychology%2C%20confirmation%20bias,occurs%20frequently%20in%20web%20searches.

Thank you for taking the time to read my entire report! 10/10 project, would do again but not procrastinate.