CSCI 104
Staff
Schedule
Homework
Labs
Exercises
Syllabus
Sections
Wiki
Resources

CSCI 104

Homework 2

- Due: See homework page
- Directory name in your github repository for this homework (case sensitive): `hw2`

Skeleton Code

Some skeleton code has been provided for you in the `hw2` folder and has been pushed to the Github repository `resources`. If you already have this repository locally cloned, just perform a `git pull`. Otherwise you'll need to clone it.

Written Portion

Problem 1 - More git Questions (6%)

In this problem, suppose we are working with a fictitious repository called `SampleRepo` (**Note: this repo doesn't exist so you can't try the commands and expect them to work**) which has a file `README.md` already committed to it. Let us now measure your understanding of the file status lifecycle in git. Please frame your answers in the context of the following lifecycle based on your interaction with the repository as specified below:

Git File Status Lifecycle

> figure courtesy of the Pro Git book by Scott Chacon

Notes:

- Parts (a) through (f) should be done in sequence. In other words, when you get to part (f), you should assume that you already executed the earlier commands (a), (b), …, (e). You **must** use the terminology specified in the lifecycle shown above, for example the output of `git status` is not accepted as a valid answer.
- For the purposes of this question, you can assume you have full access (i.e. read/write) to the repository.
- For this problem you may use online sources to look up information about `make` and Makefiles, but please cite your sources).
- **Place your answers in a file named `hw2.txt`.**

Part (a):

What is the status of `README.md` after performing the following operations:

```
#Change directory to the home directory
cd
#Clone the SampleRepo repository
git clone git@github.com:/SampleRepo.git
#Change directory into the local copy of SampleRepo
cd SampleRepo
```

Part (b):

What is the status of `README.md` and `fun_problem.txt` after performing the following operations:

```
#Create a new empty file named fun_problem.txt
touch fun_problem.txt
```

```
#List files
ls
#Append a line to the end of README.md
echo "Markdown is easy" >> README.md
```

Part (c):

What is the status of `README.md` and `fun_problem.txt` after performing the following operation:

```
git add README.md fun_problem.txt
```

Part (d):

What is the status of `README.md` and `fun_problem.txt` after performing the following operations:

```
git commit -m "My opinion on markdown"
echo "Markdown is too easy" >> README.md
echo "So far, so good" >> fun_problem.txt
```

Part (e):

What is the status of `README.md` and `fun_problem.txt` after performing the following operations:

```
git add README.md
git checkout -- fun_problem.txt
```

Also, what are the contents of `fun_problem.txt`? Why?

Part (f):

What is the status of `README.md` after performing the following operation:

```
echo "Fancy git move" >> README.md
```

Problem 2 - Makefiles (8%)

Continue your answers in `hw2.txt`.

Part (a):

Every *action* line in a makefile must start with a:

1. TAB
2. Newline
3. Capital letter
4. Space
5. It doesn't matter, any character can start an action line

Part (b):

Look at the Makefile below and answer the following question. Assume this Makefile is in the current directory, and all required files are available.

```
IDIR=.
CXX=g++
CXXFLAGS=-I$(IDIR) -std=c++11 -ggdb
```

```
LDIR =../lib

LIBS=-lm

DEPS = shape.h

OBJ = shape.o shape1.o shape2.o

%.o: %.cpp $(DEPS)
	$(CXX) $(CXXFLAGS) -c  $< -o $@

all: shape1 shape2

shape1: shape1.o shape.o
	$(CXX) $(CXXFLAGS) $^ -o $@ $(LIBS)

shape2: shape2.o shape.o
	$(CXX) $(CXXFLAGS) $^ -o $@ $(LIBS)


.PHONY: clean

clean:
	rm -f *.o *~ shape1 shape2 *~
```

Now we run the command

```
make clean
make shape1
```

Which action(s) will get called? What compiler command(s) with what exact parameters will get executed as a result of the action(s)?

Part (c):

What is the purpose of a .PHONY rule?

Part (d):

What are acceptable names for a makefile? Select all that applies.

1. Makefile.txt
2. Makefile
3. makefile.sh
4. makefile

Problem 3 - Linked List Recursion Tracing (8%)

Consider the following C++ code.

All of the points for this problem will be assigned based on your explanation, since we have full faith in your ability to run this program and copy down the answer.

```
struct Node {
    int val;
    Node*  next;
};

Node* llrec(Node* in1, Node* in2)
{
    if(in1 == nullptr) {
        return in2;
    }
    else if(in2 == nullptr) {
        return in1;
    }
    else {
        in1->next = llrec(in2, in1->next);
        return in1;
    }
}
```

Question a: What linked list is returned if `llrec` is called with the input linked lists **in1** = `1,2,3,4` and **in2** = `5,6`?

Question b: What linked list is return if `llrec` is called with the input linked lists **in1** = `nullptr` and **in2** = `2`?

To show work, you can draw a call tree or box diagram of the function calls using some simplified substitution of your choice rather than pointer values (e.g. "p3" for a pointer to a node with value 3). Submit your answers as a PDF (using some kind of illustration software or scanned handwritten notes where you use your phone to convert to PDF) showing your work and derivations supporting your final answer. **You must name the file `q3.pdf`**.

Problem 4 - ADTs (8%)

Place your answers in `hw2.txt`. For each of the following data storage needs, describe which abstract data types you would suggest using. Natural choices would include list, set, map, queue, stack, but also any simpler data types (string, int, double) that you may have learned about before.

Try to be specific, e.g., rather than just saying "a list", say "a list of integers" or "a list of names (strings) and a GPA (double)". If you specify a map please describe what the key and value will be. Also, please give a brief explanation for your choice: we are grading you at least as much on your justification as on the correctness of the answer. Also, if you give a wrong answer, when you include an explanation, we'll know whether it was a minor error or a major one, and can give you appropriate partial credit. Also, there may be multiple equally good options, so your justification may get you full credit.

1. a data type to store the text of the steps of a recipe for how to bake a cake
2. a data type that stores all the TV station identifications (e.g. `KABC`, `KNBC`, etc.) so we can ensure new stations don't reuse the same identification.
3. a data type that stores what players (assume names are unique) are on each team (given by the team name) in a league and allows quick lookups of all the players on a team as well as if a given player is on a particular team (i.e. given a team name and player name, we can quickly ascertain if that player is on that team).
4. a data type that associates a file extension (e.g. `cpp`, `pdf`) with the possible programs that are able to read/open that kind of file

Problem 5 - Class Structures and OO Design (10%)

Read the Web Search programming problem desribed in the second half of the assignment carefully (several times), and study the provided skeleton code.

Draw an inheritance diagram (an example of an inheritance diagram is shown below) of all the classes in the skeleton code, as well as all of the classes you need to create for this assignment. Show any inheritance relationship as well as **important** composition (has-a) relationships between the classes we've given (you can ignore composition relationship of basic `string`, `int`, etc. data members) of important data members). **Before you can show composition relationships, you will need to consider what data members might need to be added to various classes (in particular the `SearchEng` and derived `Handlers`).**

Provide your diagram and answers in `q5.pdf`. You can use any drawing program you wish (hand-drawn and scanned is also fine **if you draw neatly**).

Next, provide an explanation for the following questions.

1. Why do we have an abstract `PageParser` class, and why do we have a pure virtual `parse` function inside the `PageParser` class?
2. Why does the `Handler` class have a pure virtual `process` function?
3. Consider the class hierarch/organization and list the the sequence of class function calls (i.e. who calls who) that will result from an `AND` query (e.g. `AND term1 term2`), starting from the `SearchUI::run()` until the results are computed and displayed.

Example of how to show inheritance diagrams:

Programming Portion

Problem 6 - Websearch (Inheritance, Polymorphism, Sets, Maps) (60%)

Skeleton Code

Some skeleton code has been provided for you in the `hw2` folder and has been pushed to the Github repository `resources`. If you already have this repository locally cloned, just perform a `git pull`.

- [ ] Copy the contents of `hw2` (and its subdirectories) over to a `hw2` folder under your `hw-username` repository.

Don't worry about the number of source files you see. Many are complete and others will be short and/or repeat the same pattern of code. So the overall amount of code you have to write will not be TOO large. The main task is to understand how all the classes fit together and choose the appropriate ADTs in the `SearchEng` class. Read through the contents of this description a few times (even if it seems long).

Overview

This problem will have you implement a **search engine** (mimicing a toy version of Google or the like) that handles **webpages** stored in an index file, from which **parsers** for different file extensions can pull out useful information such as *text* and *incoming/outgoing links* to other webpages. We will then implement a text/menu-based user interface that provides **command handlers** for various commands which then calls into the **search engine** to carry out the command.

As you will see, all of this will require using quite a lot of the object-oriented design principles along with many of the ADTs / data structures you are learning about. You will use C++ STL `vector`, `map`, `set`, etc.

We may build off of this project in a future homework. We will not provide solutions so anything you do not get working in this HW will need to be fixed in the future. So PLEASE work hard to complete this homework.

In addition, we provide you a lot of code and structure so that you can see "good" examples of object-oriented design. Again, recall that your ability to read and understand others' code is on par with being able to write your own.

We also heavily utilize inheritance and polymorphism. Understanding how they work and the benefits of using them is a key learning outcome of doing this assignment. So please spend some time understanding and considering the given design.

At a high level, a search engine is based on the following components:

1. A real search engine would use a *crawler*, which retrieves pages from the web. You may write a simplified crawler as part of a later assignment. For now you will use a single file (aka index file) that specifies the names of the all the files/pages that should be searchable.
2. A component that parses the web page files given in the index file to extract the relevant information, such as text, links, etc. You will do this here for an extremely stripped-down version of Markdown (`.md` files) or raw text (`.txt` files), though we will use OO principles to allow for alternate formats to be parsed in the future.
3. A database (or search engine) component that stores information from parsed pages and allows for search queries by providing quick lookup of all the pages that contain the words/terms of a query.
4. A user interface component to accept commands for performing queries, displaying pages, or displaying the links of a page.
5. A ranking algorithm that takes all results and puts them in an order of relevance to the given query. You may also implement this in a later assignment.
6. Optionally, a way for users to log in so that the engine can learn about the preferences of an individual and output more relevant results. You might do that in a later assignment as well.

For this assignment, we focus on parsing and simply returning all answers to a query, without worrying about ranking or user customization. Notice, however, that you want to keep an eye on making your code well documented and **extensible** by using good object-oriented principles (encapsulation, loose-coupling, inheritance/polymorphism, and appropriate distribution of responsibilities, etc.) in case we add to it later. (That said, you will also be allowed to rewrite your code later.)

Parsing Web Pages

Your first challenge is to complete a simplified MD parser. We want our search engine to be able to support alternate file formats (TXT, MD, HTML, etc.) so we created an abstract `PageParser` class with a parse method.

```
void parse(std::istream& istr,
           std::set<std::string>& allSearchableTerms,
           std::set<std::string>& allOutgoingLinks);
```

In general, we want to parse files and find all the searchable terms. To simplify our definition of *searchable terms*, we will consider text consisting of letters, numbers, and consider all other characters as **special characters**. The interpretation is that any special character (other than letters or numbers) should be used to separate words, but numbers and letters together form words (aka “terms”). For instance, the string `Computer-Science, 104 is really,really5times,really#great?I don't_know!` should be parsed into the search terms: “Computer”, “Science”, “104”, “is”, “really”, “really5times”, “really”, “great”, “I”, “don”, “t”, “know”. Thus, during parsing, any contiguous sequence of alphanumeric characters form a search term. All other characters (special characters) will be used to split search terms and, for the sake of searching, can be discarded. In addition, you may want to convert searchable terms to a standard (canonical) form so that a search for `computer` would match a webpage containing `Computer`. We have provided some functions in `util.h/cpp` that can help you convert to a standard case.

TXT File Parsing

We have provided an implementation of a `.txt` file parser (`txtparser.h/.cpp`) that you may **use for reference** when completing the following MD parser. We assume `.txt` file can contain no hyperlinks to other pages, so we only need to parse the text for search terms.

Markdown Parsing

You should complete the derived MD parser class in `md_parser.cpp` that implements the `parse` function to parse a simplified MarkDown format. If you are unfamiliar with Markdown you may like to read this tutorial and especially the links section. We will only support normal text and links in our Markdown format and parser. In addition to text, you should be able to parse MD links of the form `[anchor text](link_to_file)` where `anchor text` is any text that should actually be displayed on the webpage and contains searchable terms while `(link_to_file)` is a hyperlink (or just file path) to the desired webpage file. A few notes about these links:

- The anchor text inside the `[]` could be anything, except it will not contain any `[`, `]`, `(`, or `)`. It should be parsed like normal text described in the previous paragraph
- A valid link will have the `(` immediately following the `]`, with no characters between. If that is not the case, then the text is not a link.
- You may assume the `link_to_file` text will not have any spaces and should be read as a single string (don't split on any special characters).
- There may be text immediately after the closing `)`. You should just treat it as a new word.
- Text in parentheses that is NOT preceded immediately by `[]` should NOT be considered as a link but just normal text. So in the text: *"ArrayLists (aka vectors) support O(1) access"*, `aka vectors` and `1` should be considered normal text and not a link.

The goal of the parser is to extract all unique search terms and identify all the links (i.e. all the `link_to_files`) found in the `(...)` part of a link and return them in the `allSearchableTerms` and `allOutgoingLinks` sets that were passed-by-reference to the function.

If the contents of a file are…

```
[Hello  ] world[hi](data2.txt)bye. (Table, t-bone) steak.
```

…then `allSearchableTerms` should contain: `Hello`, `world`, `hi`, `bye` `Table`, `t`, `bone`, `steak`. In addition, `allOutgoingLinks` should contain just `data2.txt`. Note that you can return the words in any normalized case you like that would make case-insensitive searching easier.

You may implement the Markdown parser as you see fit. However, we recommend you consider using a finite state machine (FSM) approach to read the file character by character and use "states" to determine how to process/handle that character and whether text is a normal term, a link, etc. The diagram below shows a potential FSM for parsing Markdown. Here we assume we read 1 character (i.e. `c`) at each iteration until we reach the end of the file and process `c` as well as use it to transition between states. We can use the `isalnum` function from the `cctype` library in C++ to check whether a character is a valid character for a search term. In addition, we assume we maintain two strings: `term` and `link` where we can append characters until we are ready to split and start a new term/link.

MD Parsing FSM

In addition to parsing search terms, the parsers will implement a `display_text` function to generate a displayable text string. This function strips out URL/links from the text contents of a file and only shows the anchor text. For example, if the `md` file contains:

```
[Hello ] world[hi](data2.txt)bye. (Term)
```

would be displayed as:

```
[Hello ] world[hi]bye. (Term)
```

Edge cases for MD Parsing

Example 1: The below is an ill-formed link. We will allow parsing to continue, but no link would be returned/added (i.e. the set of outgoing links would be empty). So **note** that empty strings for search terms and links should just be ignored.

```
[empty url afterwards]()
```

Example 2: For the below, all the text after `[` would be parsed as terms but no link would be generated.

```
[there is no closing square bracket
```

Example 3: For the below, everything after the `(` can be considered part of the link, though obviously it would be wrong. Essentially, if the user makes this mistake, they should not have any expectation of our parser to recover.

```
[some text](there-is-no-closing-parentheses
```

Example 4: The example below happens all the time in web pages. Still list the link. The linked page/file just doesn't exist on the server.

```
[some text](url-to-nonexistent-file)
```

Example 5: The example below if fine. There are just no terms to parse, but the link is valid.

```
[](some-file)
```

Index files

Your actual search engine application will need a list of all the webpages to parse. This would normally be done with some kind of web crawler application but for now we will just provide a text file (called an **index** file) that will contain the names of all the webpage files you need to parse and be able to search. Here is a sample index file that assumes the pages exist in a subdirectory named `test-small` underneath your hw2 folder.

File: `test-small/index.in`

```
test-small/pga.md
test-small/pgb.md
test-small/pgc.md
test-small/pgd.md
test-small/pge.md
test-small/pgf.md
test-small/pgg.md
test-small/pgh.txt
```

The contents of `index.in` are the file names of the web pages themselves, one per line. Each web page is stored in its own file. Your program should then read in all the web pages whose file name was listed in the index. There will be no format errors in the index file other than possibly blank lines, which you should just skip and continue to the next line. If any file cannot be opened you may output an error message but should continue to the next file and try to parse it.

We recommend that you store the index file and the other webpage files in a subdirectory just to keep your code and data files separated. We have already done this in the `hw2/test-small` folder provided via the `resources` repository.

The index file will be passed via the command line to your application:

```
$ ./search-shell test-small/index.in
```

Note: You may assume that file/page names **are case-sensitive** and may not contain spaces.

WebPage Class

After parsing a webpage we need to store the data and prepare it for searching and tracking link relationships. We have created a `WebPage` class for this. The code is provided in files `webpage.h/cpp`. You'll want to understand the data that these objects store and track so you can use and create them when parsing.

The function `all_terms()` should return all individual searchable terms that the parser found.

Outgoing links are those that occur in the current webpage (i.e., they point from "this" webpage to some other page). These are all known immediately after parsing. However you should also store and track (for future assignments) the incoming links which are the page (file) names that link to “this” webpage. You’ll likely need to add these little by little as you parse more pages. If, as you are parsing a page you encounter a link to another, you may need to create a `WebPage` object for that linked page before you actually parse that page. So you’ll need to keep track of whether or not pages exist and ensure you don’t make two WebPage objects for the same page. **Note:** We won’t use the incoming and outgoing links directly (other than printing them out) in this assignment but we will test your code to ensure you are identifying and adding them correctly. We just wanted you to parse and store them now so you don’t have to add that later on in the next assignment. These links could form an important part of a future homework.

As an example, if a page named `test-small/pg1.md` has the following contents

```
[another link1](test-small/pg2.md) [link3](test-small/pg3.md)
```

then `test-small/pg1.md` would have an **outgoing link** to `test-small/pg2.md` and `test-small/pg3.md`. And they in turn, would have one **incoming link** from `test-small/pg1.md`

Create a Command-line UI for Searching and Displaying Page Info

Separation of UI and Search operations

We can imagine that there may be many different interfaces to the primary web search engine. We could have a web-based interface, a stand-alone GUI application, or (as we will have for this project) a terminal-based, text interface. Because we should NOT couple the search abilities and the UI for accessing it, we will split these into different classes. You will implement a `SearchUI` and `SearchEng` class. The `SearchUI` provides a text interface that allows user to enter commands to search for and retreive webpage information and a `SearchEng` class that implements these basic operations regardless of the interface.

Search UI and Commands

In `searchui.h/cpp`, we will implement the main user interface logic which will use the terminal and perform text-based queries via some form of `std::istream/std::ostream`. The `SearchUI` class is complete but makes references to other classes in the `cmdhandler.h/cpp` and `searcheng.h/cpp` files that you will need to write.

You may assume all commands will be on a single line and NOT span multiple lines. We have defined an enumeration that will be used to return status from command handling.

- **HANDLER_OK**: The command was able to complete and processing should continue.
- **HANDLER_ERROR**: An error occurred but processing should continue.
- **HANDLER_QUIT**: The `QUIT` command was entered and the program should exit.

Menu of Commands

- `QUIT` should quit the program by returning **HANDLER_QUIT**.

- `AND word1 word2 ... wordN`: the user wants to see all the pages that contain *ALL* of the given words. The number of words here can be arbitrary. There will always be at least one whitespace between each element (`AND` and any of the search term(s)). If no terms are provided, just return an empty set. If only one term is provided, return all the pages containing that term. **No errors may occur for an AND command**.

- `OR word1 word2 ... wordN`: the user wants to see all the pages that contain ANY (at least one) of the given words. The number of words here can be arbitrary. If no terms are provided, just return an empty set. If only one term is provided, return all the pages containing that term. **No errors may occur for an OR command**.

- `DIFF word1 word2 ... wordN`: the user wants to see all the pages that contain word1 but do NOT contain *ANY* of the following words (i.e. find the pages that have word1 and then remove any of those pages that also contain word2, then remove any of the remaining pages that contain word3, etc.) The number of words here can be arbitrary. If no terms are provide, just return an empty set. If only one term is provided, return all the pages containing that term. **No errors may occur for a DIFF command**.

- `PRINT page`: displays a webpage using the display format described below (i.e. with links removed). **Return HANDLER_ERROR if the page does not exist**. You may need to catch an exception from the search engine and then return the HANDLER_ERROR
- `INCOMING page`: displays the number of incoming links for the specified page and then lists those page names (1 per line) that have a link to `page`. **Return HANDLER_ERROR if the page does not exist**.
- `OUTGOING page`: displays the number of outgoing links for the specified page and then lists those page names. **Return HANDLER_ERROR if the page does not exist**.

Queries should be case-**insensitive**, so if the user typed "UsC", and a page contained "USc", that page should be a match. In response to the query, you should tell the user how many pages matched his/her query, and if it was more than 0, display all the page filenames. We have provided this logic to you in a `display_hits(...)` function in `util.h/cpp`. **You MUST use it for display results of AND, OR, DIFF queries.** For commands that require a page name, you may assume the page/file name entered is case-sensitive and you do not have to worry about converting the cases.

Examples of each command are provided in the `test-small` folder under `search`. There is a sample index file and pages. By running the input of `query1.txt` you should get the output in `query1.exp`. The same is true for `query2.txt` and `query2.exp`. Please study these carefully to ensure you understand what each command should do.

Command Processing and Polymorphism

To process commands from the user, we will use a polymorphic approach to try to make each command and its processing **loosely coupled** from others AND to allow for easy addition of NEW commands/capabilities in the future that do not require modification of existing code. Rather than writing a large `if..else if..else if` style statement to check commands, we have provided you a base class `Handler` in `handler.h` and `handler.cpp`. This class defines a `handle` function which will invoke virtual helper functions (that are protected) to see if the derived class a.) can process this kind of command and if it can, b.) actually process (carry-out) the desired command. This approach uses a common design pattern for object-oriented programming called **"chain of responibility"**. A great website to read more about this and other design patterns is sourcemaking.com.

We have already written two derived classes to process the `QUIT` command and `PRINT` command in `cmdhandler.h` and `cmdhandler.cpp`. Use that as a guide for writing classes for the `AND`, `OR`, `DIFF`, `INCOMING`, and `OUTGOING` commands. You can put all the classes in the same file (`cmdhandler.h` and `cmdhandler.cpp`). If you look at the `handle()` function you will see its signature is:

```
HANDLER_STATUS_T handle(SearchEng* eng, const std::string& cmd, std::istream& instr, std::ostream& ostr);
```

Note: Most of the command handlers will simply call a corresponding function in the `SearchEng` class to carry out the actual task. (See the `PRINT` handler).

We will pass in the search engine so you have access to all of its public functionality, `cmd` is the identifier of the command (i.e. `QUIT`, `PRINT`, `AND`, `OR`, etc.), and `instr` and `ostr` are the the input and output streams from which you can read in the remaining, expected arguments for the specific command (e.g. search terms, etc.) and output the results (in this way, we can read and print result to files or to `cin`/`cout`).

We included this design approach so you can see how it can make processing cleaner. If you look at `SearchUI::run()` you'll notice the loop to process commands is very simple and straightforward. In addition, we achieve more loose coupling because now we can add support for new commands by simply writing a new derived `Handler` class and instantiating it and adding it to the chain in the `SearchUI`. Nothing in `SearchUI::run()` would need to change.

In `main()`, you will need to create these derived command handler objects and register them with the `SearchUI` object before calling `SearchUI::run()`. You **MUST** use this command handler approach rather than simple if statements for checking command inputs. Spend some time to understand how this chain-of-responsibility design works by reading and studying the code a few time and referring to the given website. Consider its benefits. If you have any questions, please talk to your instructor or TA.

You will need to complete the rest of `cmd_handlers.cpp` to implement the commands just described.

Search Engine

At the heart of this assignment is the search engine (database). To implement the search operations we have provided a `SearchEng` (Search Engine) class that can be used to store all the webpages objects and indexing data as well as actually performing the search operations and returning the appropriate `WebPages`. In order to be able to answer queries, you should use an **appropriate data structure that will allow you to know the set of web pages that contain a particular word/term**. Since you do NOT want to have to scan all webpages on each query, you should build this data structure as you parse all the webpages or update it anytime a webpage is added. **This data structure is the heart of this assignment, choose it carefully**. Again, given a term you would want to know the set of webpages that contain that term.

Also, in order to not run into memory problems, you probably do not want to store duplicates of WebPage objects, as that would duplicate huge amounts of text. Instead, depending on where you store the web pages, you may want to use set of indices (i.e. an integer index to a list) or pointers to WebPage objects. The exact choice here is up to you.

Combining Search Results

For the `AND`, `OR`, and `DIFF` commands there is great amount of commonality in the code you would write to implement these searches. Each of these commands may provide multiple search terms. Much of the code for implementing these 3 approaches is the same except for how to combine the set of webpages that have one term with the set of webpages for another term. Thus rather than repeating our code we will use polymorphism. The `SearchEng::search()` function takes in a generic `WebPageSetCombiner` pointer to use to combine two sets of webpages based on a particular strategy: AND, OR, DIFF. You will implement 3 derived classes, one for each strategy. In this way, we not only can avoid duplicate search code but we can potentially come up with additional search strategies in the future. You should implement these three derived classes in the `combiners.h/cpp` files. The appropriate command handlers should create and pass in the appropriate `WebPageSetCombiner` to `SearchEng::search()`. In `SearchEng::search()`, you will need to look up the sets of webpages that contain a particular term and then combine them by calling the virtual `WebPageSetCombiner::combine` function. These `combine` functions must run in **O(m log(m))** and NOT **O(m^2)** where m is the size of a set.

Furthermore, if we suppose there are `n` total searchable terms used over all webpages and a user performs a query with `k` search terms, and the maximum number of webpages that match any single term is `m`, then the query/search **MUST** be performed in runtime **`O(k*log(n) + k*m*log(m))`**

The `SearchEng` class is also responsible for storing the various parsers and applying them when files are read. We can register parsers for files with given extensions (e.g. `.md` or `.txt` files). You will need to track the parsers for each extension. You should assume that once registered, the `SearchEng` class *owns* these parsers and is responsible for their cleanup. Then we will also need the ability to actually read and parse files given an **index** file. The `SearchEng` class has the following two member functions for this purpose:

```
void register_parser(const std::string& extension, PageParser* parser);
void read_pages_from_index(const std::string& index_file);
```

The second function will be given the filename of an index file (which in turn contains names of all the webpage as described earlier in this section). You should read each file specified in the index file one at a time and update/add it (and its information) to your data structures.

To parse the files use the appropriate parser that was registered for each file's extension. By having a base `PageParser` the client can pass in derived implementations for MD parsing, HTML parsing, etc.

PRINT/DISPLAY a Page: `SearchEng` should also use the appropriate parser to generate the display version of a webpage. This should be implemented in the `display_page` member function which should output (to the provided `ostream`) the page's text as seen in the file except when you encounter a link. For links you should just print the anchor text in brackets, but not the contents of the parentheses link or the file name. So for the MD file example provided earlier, you'd display:

```
[Hello  ] world[hi]bye. (Table, t-bone) steak.
```

To generate this display text you can call the related page parser's `display_text` function.

You may add any **additional** member data and functions you desire to this class but do not change the given functions' interface.

Overall Application and main()

Your user interface, search engine and the main application will be initiated in `search-shell.cpp` which contains `main()`. Here you will create the search engine and search UI objects as well as the necessary parsers and command handlers.

`main()` is defined in `search-shell.cpp` and will require some modification.

Input and Output Stream Usage

To allow for automated testing we will support input from either the keyboard (`cin`) or a file of pre-written commands in a "query" file (via an `ifstream`). Similarly, output from your program can be sent to `cout` or an output file (`ofstream`). The user will indicate their desire via command line arguments. In general, the user will provide between 1 and 3 command line arguments.

1. The first command line argument will be the index file of webpages to parse.
2. The optional second command line argument will be an input file of pre-canned commands to run
3. The optional third command line argument will be an output file to save all the outputs from the commands in the input file. (Note: We cannot have an output file without using an input file)

So if the user runs the following command, then both input and output should be taken from `cin`/`cout`.

```
$ ./search-shell data/index.txt
```

Or if the user runs the following command, then input commands should be taken from the file `cmd1.in` while output should go to `cout`.

```
$ ./search-shell data/index.txt cmd1.in
```

Or if the user runs the following command, then input commands should be taken from the file `cmd1.in` while output should go to `cmd1.out`.

```
$ ./search-shell data/index.txt cmd1.in cmd1.out
```

All of our `SearchUI` functions use generic `std::istream` or `std::ostream` references so that you can pass in either `cin` or an `ifstream` and similarly `cout` or an `std::ofstream`. By passing in the correct streams from `main()` to `SearchUI::run()` the remainder of your code need not be concerned with the source of input or output.

Completing your Makefile

Also remember to complete the provided `Makefile` to ensure all your code compiles correctly. Follow the sample targets already provided and complete the the remaining targets and target for the executable `search-shell`. Again, a change in a `.h` or `.cpp` should trigger compilation of only those `.o` files that depend on that `.cpp` and related `.h` files. The executable should be recompiled if any `.o` file has been updated.

Other Requirements

- You may not use any algorithms from `<algorithm>` and that includes `set_intersection`, `set_union`, etc. Also, you may still NOT use the C++ `auto` keyword or ranged loops.
- You may not change any public interface of a provided class. You may add private helper functions as desired, and data members where needed.
- Be sure to meet the runtime requirements for `SearchEng::search()` described above
- Be sure to handle the check for and throw the appropriate exceptions listed in the header file (`searcheng.h`) documentation for various `SearchEng` member functions.

Recommended Ordering of Implementation

The files you will need to complete are:

- `cmdhandler.h` and `cmdhandler.cpp`
- `combiners.h` and `combiners.cpp`
- `md_parser.cpp` and potentially `md_parser.h`
- `search-shell.cpp`
- `searcheng.h` and `searcheng.cpp`
- `Makefile`

All other files are complete, though you'll need to read through them carefully to understand what you've been given.

We recommend the following order of implementation:

- ☐ Complete your `MDParser`. You may consider writing a separate test program (i.e. `.cpp` program with a `main()`) that creates an MDParser and parses a sample MD file, but we have also provided a set of MD parsing tests in `mdparse-tests.cpp`.
- ☐ Update/complete your `Makefile` for each `.cpp` file.
- ☐ Complete the various `WebPageSetCombiner` derived implementations (in `combiners.h/cpp`)
- ☐ Implement the remainder of the `SearchEng` class.
- ☐ Implement the derived `Handler` implementations for each UI command in the `cmdhandler.h/cpp` files.
- ☐ Complete the `main()` application in `search-shell.cpp` by creating the various objects, registering them, etc.

Testing

Small Data Set

While we will provide unit tests, you can also use the data files in the `test-small` folder to run sample queries. Examine the `.md` and `.txt` files provided and consider what AND, OR, DIFF, INCOMING, and OUTGOING commands might be useful to try to validate your implementation. To help get started we have provided two sample sets of input commands and the expected output.

```
./search-shell test-small/index.in test-small/query1.txt test-small/query1.out
```

You can then compare your output (`test-small/query1.out`) to the expected output (`test-small/query1.exp`).

You can repeat the same for `query2.txt`.

Be sure to understand the contents of the pages in `test-small` and what the commands in `query1.txt` and `query2.txt` **SHOULD do** and **why the expected outputs are what they are**, so that if your output doesn't match the expected output you can know where to start debugging.

Here is an image of the graph that the webpages in the `test-small` directory (i.e. `pga.md`, `pgb.md`, etc.) and their links create.

webpage graph

Golden Versions of the Project

So that you can more easily ensure your program behaviour matches our expected behavior, we have provided a **COMPILED, WORKING** executable of the solution. We have a version for newer M1 Macs and all other laptops running Docker: `search-shell-m1` (for M1 Macs running Docker) and `search-shell-x86` (for all other laptops running Docker).

So if you like you can run that with some inputs to see what should happen:

```
./search-shell-x86 test-small/index.in
```

Or regenerate the expected output described above by running:

```
./search-shell-x86 test-small/index.in test-small/query1.txt test-small/query1.out
```

(If you are on an M1 laptop, replace `x86` with `m1`).

Checkpoint

For checkpoint credit, commit and push your `hw-username` repo with a `hw2` subfolder that contains your `hw2.txt` and `q5.pdf` file with the answer to:

- Your solution to question 4 (ADTs). You may revise your answer in your final submission but need to show your answers and provide appropriate justification.
- Your solution to question 5 (Class Organization). This will ensure you have read and started to consider your class design and approach to the web search programming problem.

As well as:

- a working implementation of `md_parser.h/cpp` that can pass the `mdparser-tests` tests in `mdparser-tests.cpp`. To attempt to compile and run the `mdparser-tests` tests, type `make parser-tests` at the command line which will both compile AND run the tests (if the compilation succeeded). To pass the checkpoint, you'll need to run valgrind on `mdparser-tests` and ensure there are no memory errors.

- **THEN** you must submit your SHA on our Submit page linked from the Homework Page

Submission Files

Ensure you add/commit/push all your source code files, `Makefile`, and written problem files. Do **NOT** commit/push any test suite folder/files that we provide from any folder other than the `resources/hw2` repo.

WAIT You aren't done yet. Complete the last section below to ensure you've committed all your code.

Commit then Re-clone your Repository

Be sure to add, commit, and push your code in your hw2 directory to your `hw-username` repository. Now double-check what you've committed, by following the directions below (failure to do so may result in point deductions):

1. In your terminal, `cd` to the folder that has your `resources` and `hw-username`
2. Create a `verify-hw2` directory: `$ mkdir verify-hw2`
3. Go into that directory: `$ cd verify-hw2`
4. Clone your hw_username repo: `$ git clone git@github.com:usc-csci104-spring2022/hw-username.git`
5. Go into your hw2 folder `$ cd hw-username/hw2`
6. Switch over to a docker shell, navigate to the same `verify-hw2/hw-username/hw2` folder.
7. Recompile and rerun your programs and tests to ensure that what you submitted works. You may need to copy over a test-suite folder from the `resources` repo, if one was provided.