Project 2 - Search

CS 251, Spring 2024

In this project, we’ll take a *client view* of C++ data structures. We’ll become familiar with how to use two more of the fundamental data structures: sets and maps. By the end of this project, you will…

* Have implemented a document search engine that supports complex queries
* Have experience working with built-in C++ data structures (set and map)

Remember, if you get stuck for more than 30 minutes on a bug, you should come to office hours. You should also come to office hours if you have questions about the guide or starter code, even if you haven’t written any code yet.

## Search Engines

For our search engine, each web page has a URL ("Uniform Resource Locator") that serves as its unique id and a string containing the body text of the page. We will preprocess the body text, and store it in an appropriate data structure for fast query and retrieval.

Some questions we’d like to be able to answer include:

* Which pages contain the word “pointer”?
* Which pages contain either “simple” or “cheap”?
* Which pages contain both “tasty” and “healthy”?
* Which pages contain “tasty”, but do not contain “mushrooms”?
* Which pages contain “tasty”, but not “mushrooms”; or contain “simple”; and do all of the before and also contain “cheap”?

## Restrictions

Now that we’re more comfortable with C++, we have a slightly different set of restrictions:

* **Feel free to include additional standard C++ libraries, but no external libraries.**
* Don’t use structs, pointers, new, or malloc.
* Don’t add global variables. (Define “global” variables in main, and use pass-by-reference.)
* Don’t change the provided function signatures.
* Keep in mind function decomposition and program complexity.
  + As a general principle, if we see code that has more than 2 levels of explicit looping, this means there is probably a missed opportunity for function decomposition. Consider moving one or more loops into a function, and calling the function.

## Logistics

* Due:
  + Lab Check-In, inverted index diagram ([guide](#_grywfhes7104)): **Start of lab, 2/7-2/8**. Should be done prior to lab. This will be part of your lab grade.
  + zyBooks: **Tuesday 2/13, 11:59 PM**
  + Gradescope: **Tuesday 2/13, 11:59 PM**
    - The Gradescope assignment is not ready yet. We expect it to be available Monday 2/5.
    - search\_tests.cpp (your tests)
    - search.h
* Use grace tokens: <https://docs.google.com/forms/d/e/1FAIpQLSfG_XYYJT4twFRpcJeiSx5dTbeotd7m0YXWr8Dqog9YQcdojg/viewform>
  + This requires setting up a UIC Google account. If you have not yet done so, visit <https://learning.uic.edu/resources/virtual-collaboration/google-workspace/>.

## 

## Testing

One of our learning outcomes of CS 251 is being able to evaluate the correctness of your own code, especially without an autograder. Unfortunately, hoping that the code is correct is not particularly effective, nor is reading the code and thinking very hard.

The most common way to evaluate some code’s correctness is with ***tests***, which run the code and check the results. Autograders are made up of many tests – but outside of your classes, you probably won’t have an autograder, and will have to create your own tests. To practice this, starting with this project, you will write tests for your own programs. This is likely the first time you are writing tests, so we’ll give a lot of explicit guidance to start with!

We will evaluate your tests in 2 ways:

1. **Your tests will be graded.** When you submit, we will run your tests on a reference solution that tracks what sorts of “behaviors” your tests successfully test. You’ll see automated feedback on your tests, including whether your tests pass on a reference solution; and whether your tests cover the desired behaviors.
2. **The test suites that you will see are not comprehensive.** On Gradescope, we will have a small number of *hidden tests* for which you will not know your score until after the grace token due date. For this project, we will tell you what these hidden tests are checking for, so that you can practice writing your own test cases.  
     
   Essentially, we won’t directly give you feedback on your work. Instead, we’ll give you feedback on how well you check your own work.

We’re putting this in a red box because it’s so important.

**There are hidden tests on Gradescope for which you will not see your score until after we publish grades.** Instead, we will tell you what these hidden tests check for, and you will write your own tests. The zyBooks autograder will give you feedback on how well your tests cover the behavior checked by the hidden tests.

If:

- Your code passes the non-hidden tests,

**- Your tests use EXPECT\_EQ or ASSERT\_EQ correctly,**  
- Your tests pass on your code,  
- And the autograder says that your tests have good coverage,  
  
Then you will pass the hidden tests. Otherwise, you will get a surprise when we publish grades.

You might notice that the “testing” sections come before each “implementing” sections. This is intentional! We’re practicing something called “**test-driven development**,” where we write the tests before the code. This helps solidify our understanding of what we are trying to accomplish, before writing any code.

Finally, any test suite might be imperfect. If you believe you’ve managed to sneak a bug through that breaks a later function, that’s not caught by our public tests or your tests for hidden cases, let us know! We want our test suites to be as strong as possible.

To learn more about GoogleTest, which is our test framework of choice, read the [GoogleTest Crash Course](https://docs.google.com/document/d/1mIwEJ0iOqgeDRTVdvkwYisrxAYdgsrXUzwaKt9GQA4Y/edit).

Throughout this project, write all of your tests in search\_tests.cpp.

The default output window for unit tests in zyBooks is small. You will likely need to scroll up, or resize the window to view all of it.

### Running Tests or Main

We’ve provided a Makefile with a few targets that you will find useful:

* make tests
  + This default target builds the search\_tests executable. We can run all tests in this executable with ./search\_tests, or we can use another target to run more-specific tests.
* make test\_all
* make test\_cleantoken
* make test\_gathertokens
* make test\_findquerymatches
* make test\_buildindex
  + Build and run the tests for a specific method, as determined by GoogleTest’s suites.
* make search
  + Builds the search\_main executable. Won’t work until you’re done or nearly done with the project, see [Task: searchEngine](#_f6zgxkob6nsc).
* make search\_main
  + Build and run the search\_main executable.

## Tasks

### **Task: cleanToken**

This function should take in a single “word” from the text, and return a “cleaned” version to be stored in the index. To clean a token:

* Remove any leading or trailing punctuation, as determined by [ispunct](https://cplusplus.com/reference/cctype/ispunct/).
  + Punctuation in the middle of the token is not modified.
  + Some Unicode characters like curly quotes (“) are not considered punctuation by ispunct. Don’t worry about this, just follow what ispunct says.
* Return the empty string if the token does not contain at least one letter, as determined by [isalpha](https://cplusplus.com/reference/cctype/isalpha/).
* Convert the token to lowercase.

The return value from cleanToken is the trimmed, lowercase version (or empty string if the token is to be discarded).

#### **Subtask: Testing**

We encourage reading the tests and using them as examples to write your own in search\_tests.cpp.

It’s hard to fully specify program behavior in English. **The tests that we give you are part of the project description.**

Here are some examples of test cases that we’ve provided for you, which you can see in the local test suite (clean\_tokens.h), and which will run when you run the tests locally. You’ll also see the results of these tests on the zyBooks autograder.

* Tokens without any punctuation or special characters
* Tokens with punctuation at the beginning
* Tokens with punctuation at the end
* Tokens without any letters
* Tokens with uppercase letters, and possibly punctuation

The following cases will be tested in the autograder, but you will **not** see your score until we publish grades. As mentioned earlier, you will write your own tests to verify that your code performs correctly in these situations. Your tests will be autograded, and you’ll receive feedback on what behaviors they test, and what behaviors they miss.

* Tokens with punctuation at both the beginning and end
* Tokens with punctuation in the middle, as well as possibly at the ends

The following cases will not be evaluated because we added them after release. These cases were added because they would have caught bugs that other students hit later in the project. As such, we strongly encourage you to write these tests, even though they aren’t graded.

* Tokens with numbers at an end
* Tokens that are 1 letter long, like “x”

string cleanToken(string s);

**Task:** Write tests for the cleanToken method in search\_tests.cpp.  
  
These tests must be in the CleanToken suite, i.e. TEST(**CleanToken**, YourTestNameHere).

At this point, submit in zyBooks to see the output. You shouldn’t pass any of the tests, but look at the test called “cleanTokens: your coverage”.

You should see something like the following:

**Feedback:**

Tested: punctuation at both ends

Tested: punctuation in middle, and not at either end

Tested: punctuation in middle, and at start (but not end)

Tested: punctuation in middle, and at end (but not start)

Lines starting with “Tested” are behaviors that your tests cover. On the other hand, lines starting with “Missing” are behaviors that your tests do not currently cover.

The overall test will pass if you test all listed scenarios / behaviors, if your tests pass on your code, and if your tests pass on the reference solution. The tests run on your own solution as well.

#### **Subtask: Implementing**

Now that you have a complete test suite that you can run locally, you can check your code yourself.

**Task:** Implement the cleanToken method according to the above and its documentation.

Submit your code in zyBooks again, and look at the “cleanTokens: public tests” and “cleanTokens: your code”. **You should be receiving points for both.** If not, correct any bugs in your code or test suite before moving on. Many later functions depend on this one working correctly.

**You should submit after every task, and make sure that each task is fully complete and correct before moving on.** Every function you’re implementing relies on earlier functions working correctly. If you wait until you think you’re done to submit for the first time, you will probably have an unpleasant surprise when you have strange bugs.

### **Task: gatherTokens**

This function extracts unique tokens (“words”), separated by spaces, from a given piece of text, returning a set<string>.

#### **Subtask: Testing**

Testing gatherTokens is interesting, because it relies on cleanToken working correctly. We’ll rely on the cleanToken test suite for testing how gatherTokens responds to cleanToken edge cases, and focus our gatherTokens tests on gatherTokens behavior.

As before, we’ve provided a few tests for you to run locally in tests/gather\_tokens.h. The hidden test cases that you should test yourself are:

* Text with leading spaces
* Text with trailing spaces
* Text with multiple spaces between words

set<string> gatherTokens(string text);

**Task:** Write tests for the gatherTokens method in search\_tests.cpp.  
  
These tests must be in the GatherTokens suite, i.e. TEST(**GatherTokens**, YourTestNameHere).

The Colors test in tests/gather\_tokens.h may have an incorrect “expected” in the zyBooks template, missing "yellow". The correct “expected” in the Colors test is { "blue", "gre-en", "indigo", "orange", "red", "violet", "yellow" }.

#### **Subtask: Implementing**

This function begins by *tokenizing* the text: dividing it into individual tokens (“words”), which are strings separated by whitespace. You can either use code similar to what you wrote for Project 1 (using substr and find, or other methods); or you can explore using the [istringstream](https://cplusplus.com/reference/sstream/istringstream/) class in the <sstream> library. The >> operator is a more natural choice here.

For each token, clean it using the cleanToken helper function, then store it in the set.

**Task:** Implement the gatherTokens function according to the above and its documentation.

Use the tests we’ve provided and the tests you’ve written to verify that you’ve implemented it correctly before moving on.

### **Task: buildIndex**

At this point, we’re going to start working with data.

#### **Data Format**

The data files come in pairs of lines:

* The first line of a pair is a page URL.
* The second line of a pair is the body text of that page, with all newlines removed (basically text of the page in a single string).

The first two lines in a file form the first URL-content pair. The third and fourth lines form another pair, the fifth and sixth another, and so on. To view an example data file, open any of the txt files, such as tiny.txt or cplusplus.txt in the starter code.

#### **Inverted Index**

We’re searching a *lot* of data – to do this efficiently, we must choose the right data structure to store our data.

Consider the index in the back of a book, such as a textbook. A book’s index lists words and the pages on which each word appears. For example, we might look for the word “Internet”, and find that it lists pages 18 and 821. This means that the word “internet” occurs on page 18, and again on page 821. A book’s index is an example of an [**inverted index**](https://en.wikipedia.org/wiki/Inverted_index), which we can use to go from a *word* to the *locations* the word appears.

Our search engine will use an inverted index to allow for fast queries of web pages, given search terms. Therefore, we need to build one!

#### **Subtask: Lab Activity**

**Task:** Before writing code for this section, write or draw the inverted index for tiny.txt. This will be graded as a project “check-in” during your lab session on 2/7 or 2/8. You should complete this before the lab.  
  
You can complete this before finishing cleanToken or gatherTokens.

There’s an example inverted index for tinier.txt in the tests/build\_index.h.

#### **Subtask: Testing**

Testing this function is difficult, because it reads from a file, and because it depends on cleanToken and gatherTokens. Additionally, the expected inverted index may be large and difficult to type manually. We’ll take advantage of the fact that many of the behaviors that we want to test are covered by the tests we wrote earlier for cleanToken and gatherTokens, and not write as many tests for buildIndex.

The returned-by-reference index is a map from tokens (keys) to a set of URLS (values) where that key can be found. The int return value is the number of webpages that were processed and added to the index. (If the file is not found, return 0.)

If you downloaded tiny.txt from the Google Drive folder before 2/4 11:30 PM, you may have a different version than is on zyBooks. The correct version has the token green, not gre-en.

We’ve provided a single example test that uses the file tinier.txt in tests/build\_index.h. The hidden tests will check:

* That you run against tiny.txt
* When the file is not found.

**Make sure that you check both return values here**: the int return, and the set return-by-reference. The other files are too large to test without writing significantly more complex code.

int buildIndex(string filename, map<string, set<string>>& index)

**Task:** Use the inverted index you made for tiny.txt to write an additional test for this function based on the given example test. Also, test when the file is not found.  
  
This test must be in the BuildIndex suite, i.e. TEST(**BuildIndex**, YourTestNameHere).

#### **Subtask: Implementing**

This function takes in a filename as an argument, reads the content, and populates the provided map with an inverted index. Note that this function returns an int; the inverted index is returned by reference. Don’t clear the existing data in the index.

The data format is given above, and you can assume that files follow this format. Use gatherTokens to get the set of unique tokens from the text on each page. Then, for each token in the set, update the inverted index to indicate that this token is present on the page’s URL.

int buildIndex(string filename, map<string, set<string>>& index);

**Task:** Implement the buildIndex function according to the above and its documentation. Ensure that our tests and your tests pass before moving on.

### **Task: findQueryMatches**

This function uses the inverted index to find the URLs of pages that match a search query.

The sentence string argument can either be a single search term or a compound sequence of multiple terms. A search term is a single word, and a sequence of search terms is multiple consecutive words, each of which (besides the first one) may or may not be prefixed with a modifier like **+** or **-**.

To find the matches for a query, we process the search terms from left to right:

* For a single search term, matches are the pages that contain the specified term.
* A sequence of terms is a compound query, as follows:
  + A term with no modifier has its matches unioned with the results so far.
  + A term with the **+** modifier has its matches intersected with the results so far.
  + A term with the **-** modifier uses set difference. That is, matches for this term are removed from the results so far.
* Each term should be cleaned before searching.

If you would like to review set operations from CS 151, [this resource](https://www.probabilitycourse.com/chapter1/1_2_2_set_operations.php) might be helpful.

You may assume that the query sentence is well-formed, which means:

* The query sentence contains at least one search term.
* The first search term in the query sentence will never have a modifier.
* Every search term has at least one letter.
* There are no leading or trailing spaces.

#### **Examples**

Here are some example queries and how they are interpreted:

* "pointer"
  + matches all pages containing the term "pointer"
* "simple cheap"
  + means simple OR cheap
  + matches pages that contain either "simple" or "cheap" or both
* "tasty +healthy"
  + means tasty AND healthy
  + matches pages that contain both "tasty" and "healthy"
* "tasty -mushrooms"
  + means tasty WITHOUT mushrooms
  + matches pages that contain "tasty" but do not contain "mushrooms"
* "tasty -mushrooms simple +cheap"
  + means tasty WITHOUT mushrooms OR simple AND cheap
  + matches pages that match   
    ((("tasty" without "mushrooms") or "simple") and "cheap")
  + In order…
    - Start with all the pages that contain "tasty"
    - Take out all pages(set difference) that contain "mushrooms" (**-**)
    - Add all pages (set union) that contain "simple"
    - Keep only pages (set intersection) that contain "cheap" (**+**)

#### **Subtask: Testing**

This function is, surprisingly, much more straightforward to test, because we don’t have to call buildIndex to create an index. Instead, we can create an index ourselves! See tests/find\_query\_matches.h for our tests, which you can use as examples.

The hidden test cases that you should write tests for are:

* The first query term does not appear in the index
* A later query term, possibly with a modifier, does not appear in the index

set<string> findQueryMatches(map<string, set<string>>& index, string sentence);

**Task:** Write tests for the findQueryMatches method in search\_tests.cpp.  
  
These tests must be in the FindQueryMatches suite, i.e. TEST(**FindQueryMatches**, YourTestNameHere).

#### **Subtask: Implementing**

The C++ <algorithm> library provides [set\_union](https://cplusplus.com/reference/algorithm/set_union/), [set\_intersection](https://cplusplus.com/reference/algorithm/set_intersection/), and [set\_difference](https://cplusplus.com/reference/algorithm/set_difference/) methods that can be applied to sorted containers. These methods can be used for many more things than how we’re using them, so they’re more complex to use. Here’s an example of how to use them.

We use set\_union as an example. The input sets are A and B, and we need to have an existing result set result. We call the method as follows:

set\_union(A.begin(), A.end(),  
 B.begin(), B.end(),  
 inserter(result, result.begin()));

If A = {1, 2, 3} and B = {2, 4}, and result = {2, 6}, then after the function, A and B will be the same; and result = {1, 2, 3, 4, 6}.

**Warning:** result cannot be one of the inputs. For example, this has *undefined behavior*:

set\_union(result.begin(), result.end(),

B.begin(), B.end(),

inserter(result, result.begin()));

It probably won’t crash, but it will cause *very* confusing bugs. Don’t do it. Please.

You’re also free to write your own functions for set operations.

**Task:** Implement the findQueryMatches function according to the above and its documentation.

### **Task: searchEngine**

Finally, put the pieces together! In search\_main.cpp, we take a filename from input, and pass it to searchEngine. From here, searchEngine should:

* Print how many web pages were processed to build the index and how many distinct words were found across all pages
* Enter the command loop:
  + Prompt the user to enter a query
  + Find the matching pages, and print the URLs (one on each line)
  + If the query is the empty string, exit the command loop
* Print a closing message

The exact prompts and messages are in the sample execution.

For this function, prefer using getline instead of >> to read from cin.

void searchEngine(string filename);

**Task:** Implement the searchEngine function according to the above and its documentation.

If you are failing the searchEngine tests in zyBooks (especially the index size) you may have some issues with cleanToken or other functions. **No matter how good our test suite is, we still might have missed checking some things that are buggy!** Here’s some extra test cases that you should consider writing tests for. (These aren’t graded because we added them well after the project was released, but they would have caught bugs that other students hit. As such, we strongly encourage you to write these tests.)

**cleanToken**

* Tokens with numbers at an end
* Tokens that are 1 letter long, like “x”

At this point, you can now make search and run the entire application!

#### Sample Execution

User inputs are highlighted in **red**.

➜ ./search\_solution

**cplusplus.txt**

Stand by while building index...

Indexed 86 pages containing 1498 unique terms

Enter query sentence (press enter to quit): **vector**

Found 8 matching pages

https://www.cplusplus.com/reference/array/array/

https://www.cplusplus.com/reference/bitset/bitset/

https://www.cplusplus.com/reference/forward\_list/forward\_list/

https://www.cplusplus.com/reference/list/list/

https://www.cplusplus.com/reference/queue/priority\_queue/

https://www.cplusplus.com/reference/stack/stack/

https://www.cplusplus.com/reference/vector/vector-bool/

https://www.cplusplus.com/reference/vector/vector/

Enter query sentence (press enter to quit): **vector +container**

Found 7 matching pages

https://www.cplusplus.com/reference/array/array/

https://www.cplusplus.com/reference/forward\_list/forward\_list/

https://www.cplusplus.com/reference/list/list/

https://www.cplusplus.com/reference/queue/priority\_queue/

https://www.cplusplus.com/reference/stack/stack/

https://www.cplusplus.com/reference/vector/vector-bool/

https://www.cplusplus.com/reference/vector/vector/

Enter query sentence (press enter to quit): **vector +container -pointer**

Found 6 matching pages

https://www.cplusplus.com/reference/array/array/

https://www.cplusplus.com/reference/forward\_list/forward\_list/

https://www.cplusplus.com/reference/list/list/

https://www.cplusplus.com/reference/queue/priority\_queue/

https://www.cplusplus.com/reference/stack/stack/

https://www.cplusplus.com/reference/vector/vector/

Enter query sentence (press enter to quit):

Thank you for searching!

## 

## Grading Breakdown

|  | **Public Tests** | **Your Tests** | **Hidden Tests** |
| --- | --- | --- | --- |
| cleanTokens (10) | 7 | 2 | 1 |
| gatherTokens (20) | 17.5 | 5 | 2.5 |
| buildIndex (25) | 17.5 | 5 | 2.5 |
| findQueryMatches (30) | 21 | 6 | 3 |
| searchEngine (5) | 5 | – | – |

Additionally, there is a manually graded style and documentation component on Gradescope:

**Style**

* 2 points: Code is styled consistently; for example, using the VSCode formatter.
  + (F1, type in “Format Document”)
* 1 point: Code is reasonably styled, but there are consistent significant stylistic issues (e.g. inconsistent indentation, line length > 120, spacing, etc.)
* 0 points: No credit (e.g. entire program is on one line)

**Documentation + Commenting**

* 3 points: Code is well-documented with descriptive variable names and comments, but not overly documented.
* 1.5 points: Code is partially documented, due to a lack of comments and/or poor naming; or code is overly documented with unnecessary comments.
* 0 points: Code has no documentation or appropriate names.

## Acknowledgements

Julie Zelenski - Stanford University; Joe Hummel, PhD - Northwestern University; Shannon Reckinger, PhD - University of Illinois Chicago; Adam Koehler, PhD - University of Illinois Chicago. Modifications by Ethan Ordentlich.