Pset 6: Grop

Due: Saturday April 29, 2023 11:59 PM NHT (New Haven Time)

The (Real) Grep Program

Your assignment is to make a program that behaves similarly to the Unix `grep` program. Look up what this program does with the command `man grep` and then try a few searches to see what it outputs. For example, if I run the command `grep -R include .` we are looking for all files within the current directory (`.`) that contain the word "include". From the directory with the starter code for this pset, the result is the following one:

```
$ grep -R include ~/cpsc223/psets/pset-6-grop/starter
./DirNode.h:#include <vector>
./DirNode.h:#include <string>
./DirNode.h:#include <filesystem>
./FSTree.h:#include "DirNode.h"
./FSTree.h:#include <string>
./hashExample.cpp:#include <iostream>
./hashExample.cpp:#include <string>
./hashExample.cpp:#include <functional>
grep: ./the_grop: binary file matches
./DirNode.cpp:#include "DirNode.h"
./DirNode.cpp:#include <filesystem>
./DirNode.cpp:#include <string>
./DirNode.cpp:#include <exception>
./FSTree.cpp:#include "FSTree.h"
./FSTree.cpp:#include <filesystem>
grep: ./the_sl: binary file matches
./the_sl.cpp:#include <iostream>
./the_sl.cpp:#include <filesystem>
./the_sl.cpp:#include <string>
./the_sl.cpp:#include "FSTree.h"
./the_sl.cpp:#include "DirNode.h"
./Index.cpp:#include "Index.h"
./Index.cpp:#include <filesystem>
./Index.cpp:#include <algorithm>
./Index.cpp:#include <fstream>
./Index.cpp:#include "DirNode.h"
./Index.cpp:#include "FSTree.h"
./Index.h:#include <iostream>
./Index.h:#include <vector>
./Index.h:#include "DirNode.h"
./Index.h:#include "Entry.h"
./Index.h:#include "FSTree.h"
```

```
./Index.h:#include "Hashtable.h"
./Hashtable.cpp:#include "Hashtable.h"
./Hashtable.cpp:#include <iostream>
./Hashtable.cpp:#include <vector>
./Hashtable.cpp:#include <functional>
./Hashtable.cpp:#include "Entry.h"
./Hashtable.h:#include <string>
./Hashtable.h:#include <vector>
./main.cpp:#include <iostream>
./main.cpp:#include <fstream>
./main.cpp:#include <exception>
./main.cpp:#include <string>
./main.cpp:#include "DirNode.h"
./main.cpp:#include "FSTree.h"
./main.cpp:#include "Index.h"
```

Your "Grop" Program

Your `grop` program must perform the following steps 1. Index all the files within a directory 2. Execute (fast) search queries on the indexed directory

When you index the files you create a "database" containing information on all files within the directory. Queries are searches that the user wants you to do (such as "tell me all files that contain the word **potato**").

Reference Implementation

As with Pset 5, we have included a reference implementation called `the_grop`. Your executable should be called `grop` and should behave exactly like `the_grop`.

Start by running `./the_grop` from the command line. This program takes two command-line arguments in addition to the executable name, which is the directory to index and the name of the file to which the search results should be printed:

```
./the_grop [inputDirectory] [outputFile]
```

For example:

```
$ ./the_grop ~/cpsc223/psets/pset-6-grop out.txt
```

If the user does not specify exactly two command-line arguments (in addition to the program) name, print an error in the following format and exit the program. This is what happens in the reference implementation:

```
$ ./the_grop
usage: ./grop input_directory output_directory
```

The reference executable was compiled and will run on the Zoo but possibly not your personal computer. You might have to add the executable permission

to the file after you download it. You can do so with the command `chmod +x the_grop`.

Indexing Files

Since we will run several queries per execution of our `grop` program, it pays off to examine the whole file directory and annotate information so that future requests can be answered quickly. You must explore all files and store all words, so that whenever the user asks for a word you can quickly list all files (and line numbers within those files) that contain that word. In conceptual terms, your index should be like the index in the back of a book: given a word, on what page(s) does it appear?

The starter code will help you explore the file system (see description below) and build your index. We have also provided a header file, `Hashtable.h`, which includes an interface that you might find useful to implement to store your index.

For this assignment, our main interest is *time* efficiency. Thus, it is ok to use data structures that use a considerable amount of space. Of course, this does not mean that you are free to needlessly waste memory. In practical terms, be careful about how you store all this data. You might find that your first strategy works fine on smaller directories, but runs out of memory on larger ones. Your solution will be evaluated in part based on how quickly it indexes some large datasets. Specific requirements will be released as data sets are curated and released, but to give you a ballpark your program must index a data set of just over 1GB in under 10 minutes on the Zoo. If at any point in the indexing process a subdirectory cannot be found or you don't have permissions to open a file, simply output "Could not build index, exiting." and exit the program.

Starter Code: `FSTree` and `DirNodes`

These classes model the file system itself. They are implemented for you and should not be modified.

`FSTree`

We use a file-system tree to represent directories, subdirectories, and files. The data structure we use is an n-ary tree, so-called because a node of the tree could have any number of children. The main usage of this class is to help you navigate through folders and directories inside a computer.

An `FSTree` is an n-ary tree consisting of `DirNodes`. The `FSTree` class has the following public functions: * `FSTree(string root_name)` * Constructor. Creates an `FSTree` from the directory named `root_name` * `bool is_empty()` * Reports whether the tree is empty * `void burn_tree()` * Destroys the tree and deallocates allocated memory * `DirNode *get_root()` * Returns the root of the tree

> **Note**: the `get_root` function is a violation of the `FSTree` abstraction! As we did by making the `Node` type public in the last pset, we'll forgive this generally-bad programming pattern to avoid introducting too much complicated C++ nonsense. In this case, we'll use `get_root` to help us traverse the tree, allowing us to "do something" in each directory. We encourage you to consider how you might implement `grop` *without* violating abstraction.

An `FSTree` has only one private attribute: `DirNode *root`. The parameterized constructor (listed above) has only one parameter (a string denoting the location of the directory it should explore). If there is some error when opening files it will exit on its own. Make sure that the file system does not have a cycle when creating a `FSTree`! If so, the program will loop forever.

If all goes well, the root of the `FSTree` will be a `DirNode` containing the file information that we want. There are several private helper functions as well, just like we saw with AVL Trees. You shouldn't need them to do this assignment, but they're there for completeness.

To help you in understanding `FSTree`s, we included a program called `the_sl.cpp` for your reading and executing pleasure. It will display the contents of a directory whose name you provide at the command line.

`DirNode`

The `DirNode` class is a building block for the `FSTree` class. It is our representation of **folders**. Each `DirNode` instance has a `string name`, a `DirNode *parent`, a `vector<string>` of the names of regular files in the directory, and a `vector<DirNode *>` of subdirectories.

> **Note**: `vector` is a template class from the C++ Standard Template Library. A **template** in C++ is an extraordinarily powerful construct that lets us (among other things) parameterize types with *other types*. A key use of a template in C++ is to create generic containers (*i.e.*, types that hold objects of some other type). The `vector` class is such a generic container: when you instantiate a `vector`, you tell the compiler what type of things you'll be putting into it. In our case, a `DirNode` has one `vector<string>` and one `vector<DirNode *>`.
>
> See the description of the Hashtable interface below for more information about templates and vectors.

`DirNode` contains the following public functions (you can find the complete list of all functions including the private ones in the header file `DirNode.h`): * `bool has_subdirectory()` * returns true if and only if there are subdirectories in the current node * `bool has_files()` * returns true if and only if there are regular files in the current node * `bool is_empty()` * returns true if and only if there are no files or subdirectories in the current node * `int num_subdirectories()` * returns the number of subdirectories in the current node * `int num_files()` *

returns the number of regular files in the current node * `string get_name()` * returns the name of the current directory * `fs::path get_path()` * Returns the path to this directory *relative to the root of this tree.*

> **Note**: `fs::path` is a type from the `<filesystem>` header file, which represents a path to a file. There are many useful things you can do with a `path` that you cannot easily do with a `string`, such as compute them relative to another path, extract the file name and extension, canonicalize it (*i.e.*, `/foo/../bar//baz` could become `/bar/baz`), *etc.* A `path` is also platform-independent, meaning that you do not need to worry about the path separator on your platform to use a `path` (for example, on Windows the path separator is '\', not '/').
>
> A `path` is implicitly convertible to (and from) a `string`, so you can use a `path` variable anywhere you'd use a `string` variable (*e.g.*, during output).

- `DirNode *get_subdirectory(int n)`
  - returns a pointer to the nth subdirectory
- `string get_file(int n)`
  - returns the name of the nth regular file
- `DirNode *get_parent()`
  - returns a pointer to the parent directory node

> **Note**: You should not modify anything in `FSTree` or in `DirNode`! In fact, we will compile your submission with the version of these files provided to you as starter code (if there are any corrections that must be made, we'll use the latest version of these files that we release).

Query format

Once you have indexed all files, your program should print "`Query?`" and wait for input from the user (note that there is a space after the question mark). Your program must then respond to the user entering a query string. That string is what we need to search. Your program should list all the lines in the indexed files where the string appears. This is a case-sensitive search, so "CPSC223" and "cpsc223" are two different strings. Your program must repeatedly prompt the user for a query until the user enters "`@q`" or "`@quit`" or types `ctrl-D`. At that time, it must print "`Goodbye! Thank you and have a nice day.`" and exit the program with status code `0`.

Query: What's a Word?

We define a word as a string that starts and ends with an alphanumeric (letter or number) character. If non-alphanumeric characters are at the beginning or end of the user's query, ignore them (non-alphanumeric characters in the middle

are kept). For example, the following queries are treated identically: * `cpsc` * `#@cpsc` * `#$cpsc$$`

If the query contains more than one word, your program must behave as if the user had typed in the words as separate queries. In other words:

```
Query? we are the champions
```

Is the exact same thing as

```
Query? we
Query? are
Query? the
Query? champions
```

Reporting Queries

Beware: most of the *messages* of your program should go to standard out (`std::cout`). This includes any interaction message (such as "`Query?`"). However, the *result* of the query should instead be written into the output file (the name of which is provided as the second parameter when you execute `grop`). We recommend you execute a couple of times the reference executable (`the_grop`) to clearly see what is reported where. For each query request you should print, for each line in which the query appears: 1. The **name of the file** in which the query word appears, with its full path relative to the directory from which you executed `grop` 2. The **number of the line** in the file where the query word appears 3. The **full line** on which the query word appears

Do not forget to put a line break after each line of results. Each item on a line is separated by a colon (`:`) from the others. For example, given a directory `lyrics` that contains text files with the lyrics to some Queen songs:

```
$ ./grop lyrics out.txt
Query? we
Query? @q
$
```

When we open `out.txt` we would see:

```
lyrics/champions.txt:5: we are the champions
lyrics/rock.txt:9: we will rock you
lyrics/champions.txt:7: we will keep on fighting
```

If a query is not found in any files in the given directory, output "query not found" (also followed by a line break in the output file) but don't exit. The order in which you report the lines and files does not matter. So, any reordering of the three lines above is fine.

Starter Code: `Index` and `Hashtable`

The `Index` class

To enable you to focus on the task at hand and not get bogged down with periphery of worrying about file parsing given the relatively short turnaround period for this assignment, we've implemented most of the `Index` class for you. The `Index` class has the following public functions, all of which are implemented for you: * `std::string &clean_string(std::string &s) const` (implemented for you) * Removes leading and trailing non-alphanumeric characters from `s`, and returns a reference to the modified string * `void query(const std::string &word, std::ostream &out) const` (implemented for you) * Performs a query for `word` to this index, printing matches to `out` * `void init(std::string root)` (implemented for you) * Creates an index of the file system relative to directory named `root` * `Index(std::string root)` (implemented for you) * Creates a new Index of the file system relative to the directory named `root` * `Index()` (implemented for you) * Creates a new, empty index

It also contains several private functions, the most important of which are: * `void build_index(DirNode *node)` (**you must implement this**) * Iterates through a directory tree, parsing any files found to store the file's words in this index * `void parse_file(const std::string &filename)` (implemented for you) * Parses all words from the file named `filename` and inserts them into `this`

The `Hashtable` class

The core of your `grop` program is the hash table. You are required to implement a chained hash table for use in this assignment. The hash table must dynamically resize itself when the load factor is larger than `max_load_factor`.

Our hash table is a **template class**. That means—just like a `std::vector`—instantiation of a `Hashtable` is accompanied by a template parameter, which appears after the type name enclosed in angle brackets. The `Key` type in our hash table is always `std::string`; the template parameter is the type of the `Value` associated with each key. For example, we might instantiate a `Hashtable` that stores values of type `string` as follows:

```
Hashtable<std::string> offices;
```

We could then add some `string`/`string` pairs to the table with the following statement:

```
offices.insert("Alan Weide", "AKW 002");
offices.insert("Ozan Erat", "AKW 010");
```

Actually, our `Hashtable` interface mimics the `std::unordered_map` interface and overloads `operator[]`, so we can be more concise:

```
offices["Alan Weide"] = "AKW 002";
offices["Ozan Erat"] = "AKW 010";
```

Most of the functions that you're required to implement are methods of the `Hashtable` class. The class has the following public functions: * `Hashtable()` (implemented for you) * Default constructor. Creates a table with size `initial_capacity` (defined in `Hashtable.h` to be `1024`). * `Hashtable(size_t tablesize)` * Parameterized constructor. Initialize table with `tablesize` buckets * `Value* insert(const Key &key, const Value &to_insert)` * Inserts the `Key`/`Value` pair into this hashtable, updating the existing value if that key is already in the table. The `insert` function returns a pointer to the inserted value. * `Value &at(const Key &key)` and `const Value &at(const Key &key) const` (implemented for you) * Returns a reference to the key's corresponding value in this table * If the key is not found in the table, the `at` function throws an exception. * We won't cover exceptions in class, so this function is implemented for you. You shouldn't need to worry about the exceptional behavior in this function; uses of it are in the pre-implemented parts of `Index.cpp`.

> **Note**: The `at` function is overloaded with a `const` version. The implementations of these two versions are essentially identical; we implement both of them to match the `unordered_map` interface (in which the overload is present for a variety of good reasons on which we can elaborate if you're curious).

- `Value *find(const Key &key)` and `const Value *find(const Key &key) const`
  - Checks if this table has a stored value for `key`.
  - Returns `nullptr` if not found.
    Note: The `find` function is overloaded with a `const` version. The implementations of these two versions should be essentially identical; we implement both of them to match the `unordered_map` interface (in which the overload is present for a variety of good reasons on which we can elaborate if you're curious).
- `Value &operator[](const Key &key)`
  - Gets the value corresponding to the provided key.
  - If the value doesn't exist, a default one is inserted into this hash table and returned (this behavior is identical to `unordered_map::operator[]`).
    Note: The `at` functions, `find` functions, and the `operator[]` overload all have similar semantics. Their implementations should be similar, differing in how they behave when the provided key is or is not present.
- `double load_factor() const`
  - Returns the load factor of this hash table
- `size_t size() const`
  - Returns the number of elements stored in this hash table

In addition to the public methods, there are several private members that

you might find useful (but you are not required to implement or use): * `void expand()` * Doubles the size of this hash table * `size_t index_of(const Key &key) const` * Computes the index in the hash table of `key` * `struct KeyValue` * A struct to store each hash table element as a key/value pair

You'll notice that there are no other private members of `Hashtable`. You have to add them!

Remember, the "backbone" of a chained hash table should be capable of fast random access—that is, it should be possible, in (amortized) constant time, to access the item at any index in the table. Moreover, since you are using chaining to manage collisions, each element of the hash table must store arbitrarily many elements. Corollary: the number of non-empty items in the "backbone" is *not* the same as the number of items in the hash table

The `std::unordered_map` Type

We mentioned above that your `Hashtable` implementation mimics the behavior of `std::unordered_map`. It does, in that we picked several functions from that interface that are particularly useful for implementing the functionality of `grop`, but it is not a complete replica. All of the functions you implement have a counterpart in the `std::unordered_map` interface, and if you are ever in doubt about the semantics of a particular function, consult the documentation for `unordered_map` for further clarification.

Testing

We have provided a reference executable, `the_grop`, which was compiled and will run on the Zoo, but maybe not your personal computer. You can compare the output of `the_grop` with your solution using the provided `gropdiff` script, which counts and reports the number of lines that are different between two files, irrespective of their order.

For the purposes of testing your implementation's efficiency, we will provide you with several databases within the next few days. The files will be made available on the Zoo, with their presence to be announced on Canvas and Ed.

Once they are relased, you will be free to copy the folders into your own directory or download them to your computer, but beware that the large one is over 1GB. So rather than copying we suggest you work on the Zoo and make a symbolic link to those files from your working directory (see below).

In the mean time, experiment by running queries on directories you know well, such as your home directory or your project directories.

Helpful tricks

Rather than copying the test files we recommend you use a symbolic link:

```
$ ln -s /path/to/test-dirs test-dirs
```

This will create a folder in your current directory called "`test-dirs`", but that folder won't (directly) contain files. Instead, it will be a "pointer" to the *original* `test-dirs` folder in the `/path/to` directory.

Your `Makefile` should also use the flag `-O3` (that's the capital letter 'O', not a zero), which turns on aggressive compiler optimizations. If there are no bugs in your compiler it won't affect the correctness of your program, but it may make your program run a little faster.

We'll test your solution by indexing the provided directories. After indexing, we will run the queries in file `test_queries.txt` included with each database. Make sure that the output of your program matches *exactly* the output files included with the databases (up to reordering of the lines in each query result). Correctness will be assessed by asserting that every line in the reference file appears somewhere in your program's output file. (We will also run additional queries that do not appear in any of the `test_queries.txt` files.)

Data Structures

Except as otherwise noted below, you **may not** use pre-implemented data structures or functions either from the C++ standard library or from third-party libraries.

To aid you in your implementation, you may use the `std::vector` type from the Standard Template Library, which is an ordered collection of items implemented as a dynamically-sized array. You are not permitted to use any other types from the standard template library.

You may also use the `std::hash` function, declared in the `<functional>` header. Because we allow the use of `std::hash` even though we have not discussed it in class, we have created a sample program that uses it (see `hashExample.cpp`). Note that the use of `std::hash` is not compulsory; you are free to implement your own hash function.

> If you implement your own hash function, you must pay close attention to the likelihood of collisions. Naive hash functions have a tendency to cause many collisions, which can dramatically affect the runtime of your querying operations.
>
> (The first hash function I ever wrote as a student was $H(x) = x \bmod 10000$. It was very bad. Do not do that.)
>
> If you use a hashing algorithm you found through an online (or offline) search, you **must** cite that source in your `LOG.md` file for this assignment.

Other Items

1. As with all of your programs, you must take care not to introduce any memory leaks.
2. It's unlikely but possible that the reference executable is not entirely correct. We'll make an announcement if there is a new version of the reference executable that you should download.
3. You are not required to fix any style violations in the starter code.
4. All unimplemented sections of code in the starter files are marked with a "`TODO`" comment. You can use the search functionality in your editor of choice to find them and confirm you didn't miss any of them (or, if you're feeling adventurous, you can use `the_grop` or even your own `grop`!).
    - You are required to add private members to the `Hashtable` class in `Hashtable.h`
    - There are nine (9) unimplemented functions in `Hashtable.cpp`; you are required to implement seven (7) of them, with the two private functions being optional but recommended
    - There is one (1) unimplemented function in `Index.cpp` that you are required to implement

5. The starter code for this pset is quite extensive. We *strongly* recommend you read through and understand the code for `FSTree`, `DirNode`, `Index`, `Entry`, and `main.cpp` in addition to thoroughly studying the `Hashtable` interface you are required to implement. If you have any questions about C++isms in the starter code, don't hesitate to ask one of us. You might not be the first person to have such a question, so check previous Ed questions, too!

Submission and Deadline

The assignment is due on **Saturday Apr 29 at 11:59 PM NHT (New Haven Time)**. We will not accept submissions after the end of Reading Period, which is **Thursday May 4 at 7:00 PM NHT (New Haven Time)**.

Submit your solution by uploading to Gradescope all files needed to compile and run your `grop` program. You should not submit executables or object files.

Acknowledgements

Most of this assignment is borrowed with permission from a course at another university. We thank those professors for their kindness in letting us use their materials.

> **Note**: Because solutions to our benefactors' version of the assignment can be found online, their names and affiliation are elided in this document. As a reminder, all work you submit in CPSC 223 must be the result of *your own* understanding of the material and it is against

course and university policy to search for solutions to assignments on the internet or anywhere else.

Happy gropping!