Project 4

oogle

Time due: 9 PM Thursday, March 14

Introduction

The NachenSmall Media Corporation, owner of the popular portal WeHaveTheBestPortalInTheWorldIfYouCanJustRememberTheURL.com, has decided to throw away the existing search engine that they're licensing (it rhymes with bugle) and instead build their own search engine in-house. Given that the NachenSmall leadership team is comprised entirely of UCLA alums, they've decided to offer the job to the students of CS32. Lucky you!

So, in your last project for CS32, your goal is to build a simple search engine that indexes the content of a collection of web pages so the user can search those pages for keywords. If you're able to prove to NachenSmall's reclusive and bizarre CEO, Carey Nachenberg, that you have the programming skills to build a simple search engine, he'll hire you to build the real WeHaveTheBestPortalInTheWorldIfYouCanJustRememberTheURL.com search engine, and you'll be rich and famous.

Anatomy of a Search Engine

Every search engine has two primary components: an *indexing component* and a *searching component*. The indexing component is fed a series of documents (e.g., web pages) and must construct a set of data structures which allow for the efficient searching of these documents. This is called indexing.

Once you have created an index, a user can use a searching component to search through the index data structures and locate documents that contain a specified combination of words. Once the index has been built, users can search over and over without requiring any re-indexing (unless there are new pages that you wish to add to the index).

One of the key goals of the query component is to make the search extremely fast; after all, who wants to wait a half an hour to find a relevant web page! Therefore, when you design your indexing and query components, you should try to use the most efficient data structures to ensure that the user can quickly search for documents.

What Do You Need to Do?

For this project you will create five classes (each will be described below in more detail):

1. You will create a class template *MyMap* which works much like the C++ STL *map* and which must use a binary search tree as its data structure. This class will allow the user to associate an arbitrary type of keys (e.g., strings containing a word) with values any other type of your choosing (e.g., collections of webpages that words appear on).

2. You will create a class *Wordbag* that can hold the contents of a web page. It must basically keep track of what words were found in a web page, and how many times each word was seen in that page.
3. You will create a class *Indexer* that has several capabilities:
    a. The user will be able to pass in WordBag objects to an Indexer object, which will incorporate every word from each WordBag into its index.
    b. The user will be able to ask an Indexer object about a word and obtain a collection of URLs of the web pages on which that word appears, along with the count of how many times the word appeared on each such page.
    c. An Indexer object must be able to save its index to a data file on disk. It must also be able to reload this index from disk and into its in-memory data structures at a later time.
4. You will create a class *WebCrawler* that will download web pages from the Internet, insert the pages' contents into WordBag objects, and then pass these WordBag objects to an Indexer object for proper indexing.
5. You will build a class called *Searcher* that allows the user to specify a string of search terms (e.g. *"smallberg cs32 ucla"*), and uses an Indexer object (which contains index data for one or more web pages) to search for relevant web pages. It will return an ordered collection of URLs of web pages that best match the search terms.

What Will We Provide?

We'll provide a simple *main.cpp* file that brings your entire program together.

We'll provide a *Tokenizer* class that can tokenize strings (i.e., break up the string into individual words). You can use this to take a web page's contents (as a string) and extract its constituent words. The Tokenizer section below has details.

We'll provide an *HTTP* class that can be used to download a web page from a web server on the Internet (e.g., from *http://reddit.com*). If you specify the URL for a page, it will download the contents of the page and place them into a string. The section on the HTTP class below has details.

The Tokenizer Class

We provide a *Tokenizer* class for you to use in your program to simplify the process of tokenizing strings. Tokenizing is the process of breaking up a string, e.g. "this isn't a test. Really!", into a list of substrings like "This", "isn", "t", "a", "test", and "Really". To do this, our Tokenizer searches for all characters that are NOT letters/numbers (e.g. spaces, commas, periods, apostrophes, etc.) and treats these as dividers. It then spits out a bunch of individual tokens (made of letters and numbers) that are separated by these dividers.

Here is the class declaration:

```
class Tokenizer
{
public:
    Tokenizer(const std::string& text);
    bool getNextToken(std::string& token);
};
```

Here's how you'd use the class:

```
void bar()
{
    Tokenizer t("This isn't a test. Really!");
    string w;
    while (t.getNextToken(w))
    {
        cout << "token: " << w << endl;
    }
}
```

This function would write:

token: This
token: isn
token: t
token: a
token: test
token: Really

You may use this class anywhere in your program where tokenization is required. This class is defined in the *provided.h* file (which we provide for your use ☺).

But I don't know how to use C++ to access the Internet!

Oh, we knew you were going to say that! Such a whiner! But wouldn't you like to learn how to write a program that interacts with other computers over the Internet? We thought so. So we're going to provide you with a reasonably functional Internet HTTP interface that is capable of downloading pages off of the Internet for you. HTTP is the protocol used by web browsers to download web pages from servers on the Internet into your browser.

When you use our interface, you don't have to worry about the details of how to communicate over the Internet yourself. Of course, if you want to see how our interface works, you're welcome to do so… and before you know it, you'll be forming your own

start-up Internet company to compete against Google[1]. Our HTTP interface's primary public function (get) is as easy to use as this:

```
#include "http.h"

int main()
{
    string url = "http://en.wikipedia.org/wiki/Bald";
    string page;  // to hold the contents of the web page

      // The next line downloads a web page for you.  So easy!
    if (HTTP().get(url, page))
        cout << page;  // prints  <!DOCTYPE html PUBLIC ...
    else
        cout << "Error fetching content from URL " << url << endl;
    ...
}
```

Note that you don't need to declare an HTTP variable. The call above looks as if it calls a function named HTTP, then calls a *get()* member function on what it returns.

A challenge when testing a program that analyzes the contents of web pages is that you have no control over those contents. Our HTTP interface lets you set up a pseudo-web of pages with URLs and contents of your choosing:

```
int main()
{
    HTTP().set("http://a.com", "This is a test page.");
    HTTP().set("http://b.com", "Here is another.");
    HTTP().set("http://c.com", "<html>Everyone loves CS 32</html>");
    string page;
    if (HTTP().get("http://b.com", page))
        cout << page << endl;  // writes  Here is another.
}
```

You call *set()* to associate a URL with a string. From that point on, calling *get()* with that URL will retrieve that string. (Once you call *set()*, *get()* will no longer retrieve pages from the real web; it will instead consult only the pages that you installed with *set()*.)

Details: The Classes You Must Write

You must write correct versions of the following classes to obtain full credit on this project. Your classes must work correctly with our provided classes.

[1] By agreeing to use our HTTP code for Project 4, this license entitles NachenSmall to a 20% cut of all profits.

MyMap

You must implement a template class named *MyMap* that, like an STL map, lets a client associate items of a key type with items of a (usually different) value type, with the ability to look up items by key. For example, a MyMap object associating students' names with their GPAs would have string as the key type and double as the value type. Your implementation **must** use a binary search tree.

Here's an example of how you might use MyMap:

```
void foo()
{
    MyMap<string,double> nameToGPA;  // maps student name to GPA

      // add new items to the binary search tree-based map
    nameToGPA.associate("Carey", 3.5);  // Carey has a 3.5 GPA
    nameToGPA.associate("David", 3.99); // David beat Carey
    nameToGPA.associate("Abe", 3.2);    // Abe has a 3.2 GPA

    double* davidsGPA = nameToGPA.find("David");
    if (davidsGPA != NULL)
        *davidsGPA = 1.5;     // after a re-grade of David's exam

    nameToGPA.associate("Carey", 4.0);    // Carey deserves a 4.0
                                          // replaces old 3.5 GPA

    double* lindasGPA = nameToGPA.find("Linda");
    if (lindasGPA == NULL)
        cout << "Linda is not in the roster!" << endl;
    else
        cout << "Linda's GPA is: " << *lindasGPA << endl;

      // now let's print all associations out in a level-order ordering
    cout << "Here are the " << nameToGPA.size() << " associations: "
         << endl;

    string name;
    for (double* GPAptr = nameToGPA.getFirst(name); GPAptr != NULL;
                                      GPAptr = nameToGPA.getNext(name))
    {
        cout << name << " has a GPA of " << *GPAptr << endl;
    }

      // The above loop would print:
      //    Carey has a GPA of 4.0
      //    Abe has a GPA of 3.2
      //    David has a GPA of 1.5
}
```

Your implementation **must** have the following interface:

```
template <class KeyType, class ValueType>
class MyMap
```

```
{
public:
    MyMap();           // constructor
    ~MyMap();          // destructor; deletes all of the trees nodes
    void clear();      // deletes all of the trees nodes → empty tree
    int size() const;  // return the number of associations in the map

      // The associate method associates one item (key) with another (value).
      // If no association currently exists with that key, this method inserts
      // a new association into the tree with that key/value pair.  If there is
      // already an association with that key in the tree, then the item
      // associated with that key is replaced by the second parameter (value).
      // Thus, the tree contains no duplicate keys.
    void associate(const KeyType& key, const ValueType& value);

      // If no association exists with the given key, return NULL; otherwise,
      // return a pointer to value associated with that key.  This pointer can be
      // used to examine that value, and if the map is allowed to be modified, to
      // modify that value directly within the map.  (The second overload enables
      // this.  Using a little C++ magic, we have implemented it in terms of the
      // first overload, which you must implement.
    const ValueType* find(const KeyType& key) const;
    ValueType* find(const KeyType& key);

      // A client uses getFirst and getNext to retrieve each association in the
      // tree, one at a time.  For each association, the method stores the key of
      // that association in its parameter (key) and returns a pointer to the
      // value item of that association.  If getFirst is called on an empty map,
      // it returns NULL.  If getNext is called after all associations have been
      // retrieved, it returns NULL.
      // These two methods MUST return items in a LEVEL-ORDER ordering.  getFirst
      // must NOT examine any node of the tree other than the root, and getNext
      // must have a running time bounded by a constant (so it must not loop or
      // recurse over many tree nodes).
    ValueType* getFirst(KeyType& key);
    ValueType* getNext(KeyType& key);
};
```

Here are the requirements for your MyMap class:

1. You **must** implement your own binary search tree in your *MyMap* class (i.e., define your own *Node* struct/class, maintain a root/head pointer, etc). You may assume that the key type of any instantiation of the MyMap template class has appropriate comparison operators (<, <=, >, >=, ==, and !=) defined for it (certainly ints and strings do).
2. You must **not** use any STL containers to implement *MyMap*, with one exception: If you wish, you may use the STL *queue* class to help you implement *MyMap*.
3. Your *MyMap* class **must** be a template class, to enable a client to map one type of item to any other type of item, e.g., a name (string) to a GPA (double), or a name (string) to a collection of the person's test scores (a vector of ints).
4. Your *MyMap* class **must** use the public interface documented above. You may add only private members to this class; you must **not** add other public members to *MyMap*.

5. Your *MyMap* class does not need to implement individual node deletion (unless you really want to) and does not need to be balanced in any way (unless you're masochistic).
6. If a user of your class associates the same key twice (e.g., "David" → 3.99, then "David" → 1.5), the second association must overwrite the first one (i.e., "David" will no longer be assocated with 3.99, but will henceforth be associated with 1.5). There **must** be at most one mapping for any key.
7. Your *MyMap* class's *getFirst()* and *getNext()* methods **must** return items in **level order**. *getFirst()* must not examine any node of the tree other than the root node. *getNext()* **must** run in constant time (i.e., its runtime must **not** be proportional to the number of nodes in the tree).
8. If the user constructs a *MyMap* object or calls *clear()* or *associate()* on a *MyMap* object, and subsequently, without an intervening call to *getFirst()* on that object, calls *getNext()* on that object, then the behavior of the program is undefined by this spec: Your program may perform any behavior it likes, including crashing or skipping items. In other words, traversing through a *MyMap* object has undefined behavior if you don't start the traversal by calling *getFirst()* or if after starting a traversal, you add now associations and try to continue with that traversal instead of starting a new one.
9. You do not need to implement a public copy constructor or assignment operator for *MyMap*. To prevent incorrect copying and assignment of *MyMap* objects, these methods can be declared private and left unimplemented.
10. Your member functions must **not** write anything out to *cout*. They may write to *cerr* if you like (to help you with debugging).

WordBag

The *WordBag* class is responsible for taking in a string, breaking it into its constituent parts (i.e., words), and allowing a client to then retrieve each distinct word and the number of times it occurred within the string, one at a time. Here's the required public interface of the WordBag class:

```
class WordBag
{
public:
    WordBag(const std::string& text);
    ~WordBag();
    bool getFirstWord(std::string& word, int& count);
    bool getNextWord(std::string& word, int& count);
};
```

When you construct a new WordBag object, you pass in a string. (For this project, that string will generally be the contents of a webpage, e.g. *"<html>This is a web page</html>",* which was presumably already retrieved from the Internet. The constructor must break the string up into its constituent words, discarding all whitespace and punctuation marks. Our definition of *word* is a contiguous sequence of letters and/or digits, so "Hello there!" is two words ("Hello" and "there"), "don't" is two words ("don"

and “t”), "CU L8R b-bye." is four words ("CU", "L8R", "b", and "bye"). Fortunately, the *Tokenizer* class we provide for you (see below) takes care of this parsing for you, so use it! All words must be converted to lower case by a WordBag object, so that "gobs", “GoBs”, and “GOBS” are treated as the same word: “gobs”. We provide a *strToLower* function for you as well.

After constructing a new WordBag object, a user may then call *getFirstWord()* and *getNextWord()* to obtain all of the distinct words (and each word’s count) in the string. These functions return *true* if they retrieved a word from the word bag, and *false* if there were no more words to retrieve.

For example, if the string

“<html>I’m livin’ 2 love hash tables, I say.</html>”

were passed to this function:

```
void foo(const string& webPageContents)
{
    WordBag wb(webPageContents);
    string word;
    int count;
    bool gotAWord = wb.getFirstWord(word, count);
    while (gotAWord)
    {
        cout << "The word " << word << " occurs " << count
             << " times" << endl;
        gotAWord = wb.getNextWord(word,count);
    }
}
```

the function will print these lines in some order (not necessarily this order):

The word html occurs 2 times
The word i occurs 2 times
The word m occurs 1 times
The word livin occurs 1 times
The word 2 occurs 1 times
The word love occurs 1 times
The word hash occurs 1 times
The word tables occurs 1 times
The word say occurs 1 times

To ensure that you do not change the interface to the WordBag class in any way, we will implement that class for you. But don't get your hopes up that we're doing any significant work for you here: Our implementation is to simply give WordBag just one private data member, a pointer to a WordBagImpl object (which you can define however you want in WordBag.cpp) The member functions of WordBag simply delegate their work to

functions in WordBagImpl.[2] You still have to do the work of implementing those functions.

Other than WordBag.cpp, no source file that you turn in may contain the name WordBagImpl. Thus, your other classes must not directly instantiate or even mention WordBagImpl in their code. They may use the WordBag class that we provide (which indirectly uses your WordBagImpl class).

Here are the requirements for the WordBagImpl class:

1. It **must** adhere to the specification above.
2. It must **not** use any STL containers like map, hash_map, unordered_set, list, vector, etc.
3. It **must** use a MyMap object to hold its words and word counts.
4. It must **not** access any other Impl classes that you write.
5. If the user constructs a *WordBag* object and subsequently, without an intervening call to *getFirstWord()* on that object, calls *getNextWord()* on that object, then the behavior of the program is undefined by this spec: Your program may perform any behavior it likes, including crashing or skipping items. In other words, traversing through a *WordBag* object has undefined behavior if you don't start the traversal by calling *getFirstWord()*.
6. It **must** be efficient – if the string passed to the constructor has N words, adding its contents to the word bag had better take far fewer than $O(N^2)$ steps!
7. Its member functions must **not** write anything out to cout. They may write to cerr if you like (to help you with debugging).

Indexer

So what is an index? In the context of a search engine, an index is a table that holds a bunch of words. For each word in the index, the index maintains an internal mapping of the word to a collection of UrlCount objects. Each UrlCount object contains the URL of a web page that contains the word and the number of times that word appears on that page. For example, let's assume we have three web pages (content shown below):

```
www.a.com: "<html>i like gogiberries and I hate spam</html>"
www.b.com: "<html>engineering is FUN</html>"
www.c.com: "<html>Engineering majors like fun</html>"
```

Conceptually, a (case-insensitive) index for these three sites might look like this:

```
and         → {"www.a.com", 1 time}
engineering → {"www.b.com", 1 time}, {"www.c.com", 1 time}
fun         → {"www.b.com", 1 time}, {"www.c.com", 1 time}
gogiberries → {"www.a.com", 1 time}
hate        → {"www.a.com", 1 time}
```

[2] This is an example of what is called the pimpl idiom (from "**p**ointer-to-**impl**ementation").

```
html        → {"www.a.com", 2 times}, {"www.b.com", 2 times},
              {"www.c.com", 2 times}
i           → {"www.a.com", 2 times}
is          → {"www.b.com", 1 time}
like        → {"www.a.com", 1 time}, {"www.c.com", 1 time}
majors      → {"www.c.com", 1 time}
spam        → {"www.a.com", 1 time}
```

Your *Indexer* class will maintain such an index, mapping each word to the collection of URLs for the pages in which the word was found (as well as the count of how many times that word was found in each such page).

Here's the required public interface of the Indexer class:

```
struct UrlCount
{
    std::string url;
    int count;
};

class Indexer
{
public:
    Indexer();
    ~Indexer();
    bool incorporate(std::string url, WordBag& wb);
    std::vector<UrlCount> getUrlCounts(std::string word);
    bool save(std::string filenameBase);
    bool load(std::string filenameBase);
};
```

The user calls the *incorporate()* method to add all the words in the provided WordBag argument to the index. If *incorporate()* has previously been called with the same url argument, it returns false, doing nothing. Otherwise, it updates the index by adding all the words and returns true. Incorporating a WordBag containing W distinct words to an index that already contains N words must take far less than O(WN) time, because adding one word mapping (e.g., "fun" → { "www.b.com", 1 }) to an index that already contains N words must take far less than O(N) time.

The *getUrlCounts()* method returns a vector of UrlCounts for the specified word. Each UrlCount in the vector holds a URL string and the count associated with that word in the WordBag incorporated with that URL (representing the number of times that word appeared on the web page at that URL). A URL string must not occur more than once within the vector of matches. There is no requirement that the UrlCounts in the vector returned by this method be in any particular order. If the word was not in any of the incorporated WordBags, the *getUrlCounts()* method returns a vector with no elements. The method must be case-insensitive, so it must return the same result for, e.g., "zits", "ZITS", or "ZitS" as the word argument. If your index contains N words, your *getUrlCounts()* method **must** return a vector of UrlCounts in far less than O(N) time.

The *save()* method saves the index that an Indexer object has built to one or more files on disk so that the index can be loaded and used at a later time. The file(s) have names beginning with the filenameBase string passed as the argument. It returns true if the index was saved successfully, false otherwise. See below for details about this method.

The *load()* method loads a previously-saved index into an Indexer object. The previous contents of the Indexer object are discarded, so that after the load, the Indexer contains the same content that the Indexer that saved the index had at the time of the save. The file(s) containing the previously-saved index to be loaded have names beginning with the filenameBase string passed as the argument. It returns true if the index was loaded successfully, false otherwise. See below for details about this method.

Here's an example of how the Indexer class might be used:

```
void writeWordInfo(Indexer& indexer, string word)
{
    vector<UrlCount> urlCounts = indexer.getUrlCounts(word);

    if (urlCounts.empty())
    {
        cout << word << " was not found in the index." << endl;
        return;
    }
    for (int i = 0; i < urlCounts.size(); i++)
        cout << word << " appears " << urlCounts[i].count
             << " times at " << urlCounts[i].url << endl;
}

bool testIndexer()
{
    const string INDEX_PREFIX = "C:/Temp/myIndex";

    Indexer indexer;

    WordBag wb1("<html>i like gogiberries and I hate spam</html>");
    indexer.incorporate("www.a.com", wb1);

    writeWordInfo(indexer, "I");
        // writes  I appears 2 times at www.a.com

      // save the index as file(s) whose names start with prefix
    if ( ! indexer.save(INDEX_PREFIX))
        return false;  // error saving the index

      // load the just-saved index into another Indexer
    Indexer indexer2;
    if ( ! indexer2.load(INDEX_PREFIX))
        return false;  // error loading the index

      // add more pages to the second index
    WordBag wb2("<html>engineering is FUN</html>");
    indexer2.incorporate("www.b.com", wb2);
    WordBag wb3("<html>Engineering majors like fun</html>");
    indexer2.incorporate("www.c.com", wb3);
```

```
    writeWordInfo(indexer2, "like");
        // writes in some order:
        //     like appears 1 times at www.a.com
        //     like appears 1 times at www.c.com

    writeWordInfo(indexer2, "smallberg");
        // writes  smallberg was not found in the index

    return true;
}
```

After the *testIndexer()* function returns, there should be one or more files on disk whose names start with "myIndex" in the folder C:\Temp. (Under Windows, the C++ file operations accept either forward slashes or backslashes as path separators, but since backslashes have to be doubled in C++ string literals, it's easier to use forward slashes: "C:/Temp/myIndex" is easier to get right than "C:\\Temp\\myIndex".)

A realistic use of the Indexer class would not save and load an index within one run of a program as the *testIndexer()* function does. Instead, one run of a program might build an index and save it to disk, and then a run of the same or a different program might load the saved index, possibly have the index incorporate more items, and do searches with it.

Your implementation **must** efficiently store its data to minimize its memory usage. For example, consider the conceptual index presented above that started with:

```
and         → {"www.a.com", 1 time}
engineering → {"www.b.com", 1 time}, {"www.c.com", 1 time}
fun         → {"www.b.com", 1 time}, {"www.c.com", 1 time}
...
```

Notice that if we actually implemented the index this way, it would contain multiple copies of each URL string. For example, in the full example, the URL string *"www.a.com"* is duplicated 7 different times in the mappings for 7 different words: “and”, “gogiberries”, “hate”, “html”, “i”, “like” and “spam”. For a realistic use of the index, storing many copies of many long URL strings is a waste of space, since we can do better: Represent each URL string with an integer. Integers typically take up as much memory as only four characters, much less than a typical URL string. To use this approach, we might use three map data structures:

urlToId: Associates each URL string to an integer ID number:
"www.a.com" → 100
"www.b.com" → 200
"www.c.com" → 300

idToUrl: Given an ID number, tells us the URL string associated with it:
100 → "www.a.com"
200 → "www.b.com"
300 → "www.c.com"

wordToIdCounts: Replaces the lengthy URL strings with their numeric IDs:

```
and         → {100, 1 time}
engineering → {200, 1 time}, {300, 1 time}
fun         → {200, 1 time}, {300, 1 time}
gogiberries → {100, 1 time}
hate        → {100, 1 time}
html        → {100, 2 times}, {200, 2 times}, {300,2 times}
i           → {100, 2 times}
is          → {200, 1 time}
like        → {100, 1 time}, {300, 1 time}
majors      → {300, 1 time}
spam        → {100, 1 time}
```

These index data structures are much more space efficient than one that duplicates many URL strings. We store each URL string only two times (once in urlToId and once in idToUrl), regardless of how many words refer to a given URL. Our wordToIdCounts data structure stores space-efficient integer values like 100, or 200 to represent each URL. When needed, the actual URL string can then be looked up in the idToUrl data structure.

Your Indexer class **must** use an efficient mapping approach that does not duplicate URL strings more than necessary. If the number of URL strings your index contains is U, then adding or looking up a URL string or ID number in a data structure **must** take far less than O(U) time.

HINT: Be careful here. If you assign integer values to each URL in numerical order, then as you add new items to an *idToURL* data structure, you'll end up creating a very unbalanced tree that will substantially slow down your implementation! See if you can figure out how to avoid this pitfall.

If your Indexer implementation uses several data structures like this, then it's probably easier to implement *save()* and *load()* using several files rather than one. For example, you might implement Indexer's *save()* method by having it call a (non-member) template function that saves a MyMap object:

```
template<typename KeyType, typename ValueType>
bool saveMyMap(string filename, MyMap<KeyType,ValueType>& m)
{
    ...  // Save contents of m to a file
}

bool IndexerImpl::save(string filenameBase)
{
    return saveMyMap(filenameBase + ".u2i", urlToId)  &&
           saveMyMap(filenameBase + ".i2u", idToUrl)  &&
           saveMyMap(filenameBase + ".w2ic", wordToIdCounts);
}
```

If a client passed "C:/Temp/myIndex" to *save()*, the index would be stored in three files: C:\Temp\myIndex.u2i, C:\Temp\myIndex.i2u, and C:\Temp\myIndex.w2ic.

The File I/O section below has detailed implementation tips for *save()* and *load()*.

As with the other classes you must write, the real work will be implementing the auxiliary class IndexerImpl in Indexer.cpp. **Other than Indexer.cpp, no source file that you turn in may contain the name IndexerImpl.** Thus, your other classes must not directly instantiate or even mention IndexerImpl in their code. They may use the Indexer class that we provide (which indirectly uses your IndexerImpl class).

Here are the requirements for the IndexerImpl class:

1. It **must** adhere to the specification above.
2. It must **not** use the following STL containers: *C*, unordered_*C*, or hash_*C*, where *C* is map, multimap, set, or multiset. It **may** use other STL containers.
3. It **must** use MyMap objects to hold its index data structures requiring efficient lookups.
4. It must **not** access any other Impl classes that you write.
5. It **must** be as algorithmically efficient as possible given these constraints.
6. Its member functions must **not** write anything out to cout. They may write to cerr if you like (to help you with debugging).

WebCrawler

The *WebCrawler* class is responsible for downloading web pages from the Internet and adding their content to an Indexer object. The idea is that the user will first add thousands or millions of URLs to a WebCrawler object. The object would then crawl these URLs, download each web page, extract each page's words, and add these words to a master index (Hint: held as a private member of type Indexer). Once the WebCrawler object had finished crawling all of the specified URLs, the user can then save the index to a file on the hard drive for later use.

Here's the required public interface of the WebCrawler class:

```
class WebCrawler
{
public:
    WebCrawler();
    ~WebCrawler();
    void addUrl(std::string url);
    int getNumberOfUrls() const;
    void crawl(void (*callback)(std::string url, bool success));
    bool save(std::string filenameBase);
    bool load(std::string filenameBase);
};
```

After creating a WebCrawler object, the user may add URLs for crawling using the *addURL()* method. This method must simply store its argument URL in some sort of

container (and not crawl it) until such time that the user of the class calls the *crawl()* method.

The *getNumberOfUrls()* method returns the number of URLs that have been added to the WebCrawler object by *addURL()* but have not yet been processed by *crawl()*.

The *save()* method saves the index built by a WebCrawler object to one or more files on disk so that the index can be loaded and used at a later time. The file(s) have names beginning with the filenameBase string passed as the argument. It returns true if the index was saved successfully, false otherwise.

The *load()* method loads a previously-saved index into a WebCrawler object. Whatever was previously indexed by the crawler is discarded, so that after the load, the crawler contains the same content that the crawler that saved the index had at the time of the save. The file(s) containing the previously-saved index to be loaded have names beginning with the filenameBase string passed as the argument. It returns true if the index was loaded successfully, false otherwise.

The *crawl()* method is responsible for iterating through all of the URLs added to a WebCrawler object, and does the following for each:

1. Connect to the website and download the web page at the specified URL
2. If the download was successful:
   a. Place the web page into a WordBag object
   b. Incorporate the WordBag into the WebCrawler's Indexer.
3. Call a "callback" function, provided by the user via a function pointer, to report the status of the web page download and incorporation into the index.

The callback function technique is one way to let a client customize the task performed by a function. In our case, we want a function of ours to be called once per URL during the *crawl()* method's processing of the URLs. If it was always going to be the same function to be called, say *f1()*, *crawl()* would be written to just call it:

```
void crawl()  // an example of how crawl is NOT specified
{
    for each URL to process
    {
        ...
        f1(that URL);
        ...
    }
}
```

But what if we wanted to have one call to *crawl()* call our function *f1()* for each URL, another call to *crawl()* call *f2()* each time, etc.? Then *crawl()* could be designed to take a parameter that indicates which function (*f1()*, *f2()*, etc.) to call when the time comes:

```
void crawl(placeholder f representing a function)
{
    for each URL to process
    {
        ...
        f(that URL);
        ...
    }
}

void foo()
{
    crawl(f1);  // Have f in crawl represent f1
    crawl(f2);  // Have f in crawl represent f2
}
```

We call *f1()* and *f2()* in this context *callback functions*. There is no actual function named *f()* that *crawl()* calls; instead, when *crawl()* appears to call *f*, it's really calling the function supplied as an argument to *crawl()*.

So how is this done? When we pass *f1()* to *crawl()*, we're really passing a pointer to *f1()*. The declaration of the parameter *f* indicates what the signature of that function must be, i.e., what parameter types and return type it must have. When *crawl()* appears to call *f*, it really calls the function that *f* points to. Notice that this technique is not completely flexible: All callback functions passed to *crawl()* must have the same signature.

In the case of *crawl()*, this spec actually requires the callback function passed to it to have a void return type and to take two parameters: the URL that was processed and the success or failure of the attempt to download the page at that URL and incorporate its words into the WebCrawler's Indexer. The prototype of *crawl()* is:

```
void crawl(void (*f)(std::string url, bool success));
```

This says the parameter *f* is first and foremost a pointer. To what kind of thing? To a function taking a string and a bool, and returning void. We can pass any function like that to *crawl()* as its callback function. Let's see:

```
// reportStatus() is our callback function – it will be called by a
// WebCrawlerImpl every time another page is retrieved and indexed.
// It simply writes the status of each URL processed.  A more
// sophisticated callback function might update a progress bar.

void reportStatus(string url, bool success)
{
    if (success)
        cout << "Downloaded and indexed the page at " << url << endl;
    else
        cout << "Unable to download the page at " << url << endl;
}

bool testWebCrawler()
{
    const string INDEX_PREFIX = "/Users/fred/index";
```

```
    WebCrawler wc;

      // load a previously-saved index from disk
    if ( ! wc.load(INDEX_PREFIX))  // error loading
        return false;

      // specify which URLs we are to crawl and index
    wc.addUrl("http://www.yahoo.com/main");
    wc.addUrl("http://www.nytimes.com");
    wc.addUrl("http://www.symantec.com/enterprise");

      // download the specified URLs and add their contents to the
      // index, designating reportStatus as the callback function
    wc.crawl(reportStatus);

      // save the updated index to disk
    if ( ! wc.save(INDEX_PREFIX))  // error saving
        return false;

    return true;  // no errors
}
```

As with the other classes you must write, the real work will be implementing the auxiliary class WebCrawlerImpl in WebCrawler.cpp. **Other than WebCrawler.cpp, no source file that you turn in may contain the name WebCrawlerImpl.** Thus, your other classes must not directly instantiate or even mention WebCrawlerImpl in their code. They may use the WebCrawler class that we provide (which indirectly uses your WebCrawlerImpl class).

Here are the requirements for the WebCrawler class:

1. It **must** adhere to the specification above.
2. It must **not** use the following STL containers: *C*, unordered_*C*, or hash_*C*, where *C* is map, multimap, set, or multiset. It **may** use other STL containers or your MyMap class if you wish.
3. It must **not** access any other Impl classes that you write.
4. Its member functions must **not** write anything out to cout. They may write to cerr if you like (to help you with debugging). The callback function that a client passes to *crawl()* might write to cout, so calling *crawl()* may cause output to be written (which is allowable in this case), but *crawl()* must not directly itself write to cout.

Searcher

The *Searcher* class is used to search through a previously-built index created by a WebCrawlerImpl object (Hint: held as a private member of type Indexer) for web pages that match a set of user-provided search terms.

Here's the required public interface of the Searcher class:

```
class Searcher
{
public:
    Searcher();
    ~Searcher();
    std::vector<std::string> search(std::string terms);
    bool load(std::string filenameBase);
};
```

As you can see, you can initialize a Searcher object, load a previously-built index from disk, and allow the user to search for pages containing specified search terms.

The *load()* method loads a previously-saved index into a Searcher object. Whatever was previously in the Searcher object's index is discarded, so that after the load, the Searcher contains the same content that was saved to the index. The file(s) containing the previously-saved index to be loaded have names beginning with the filenameBase string passed as the argument. It returns true if the index was loaded successfully, false otherwise.

The *search()* method allows the user to search the index for documents that match a specified set of search terms. The method returns a vector of zero or more items containing URLs whose contents matched the search terms according to our specified matching algorithm detailed below.

All searches conducted by *search()* **must** be case insensitive (that is, a search for "ZITS" and "ziTs" and "zits" must all produce the same results; you must **not** assume the search terms will be all lower case). Use only the *words* in the terms string (hint: Tokenizer!). Treat a word appearing more than once in the terms string as if it appears only once.

For the purposes of this project, a given web page matches a search query (e.g., "zits milk chocolate") if at least T of the N distinct search terms are found in that web page:

> N = number of distinct words in the terms string
> T = int(0.7*N) or 1, whichever is larger.

For example, for the terms string "zits milk chocolate", N = 3, so T = int(0.7*3) = 2. Therefore, all pages containing at least 2 of the 3 specified search terms would match this query, so a page containing the word "zits" and the word "chocolate" would be a match, as would a page containing the word "milk" and the word "chocolate", or the words "zits" and "milk". Of course, a page containing all three keywords would also be a match. However a page containing only the word "zits" of the three, but not the other two words would not be considered a match. When determining whether a word contributes to the page match, the number of times the word appears on the page is irrelevant. Regardless of whether a page has one "zits" or ten "zits", the fact that it has "zits" counts only once toward the required total of T search terms for the page to match the query.

As another example, suppose the terms string is just "zits". In this case, N=1 and int(0.7*N) is less than 1, so T=1. Thus, any page containing this one search term constitutes a match.

So how do you code your search function? Well that's up to you. But here's a hint: You can take each search term, e.g. "zits", "milk", and "chocolate" and look each one up in your index. For each lookup, the index will provide you with a vector containing URLs whose pages hold that term. You can then use this data to find out if there were any web pages that held at least T of your terms.

The *search()* method will return a vector of strings, where the strings are the URLs of the matching pages. But they won't just be in any old order. Obviously, someone doing a search wants the URLs of more relevant web pages near the beginning of their search results sequence, and less relevant URLs near the end. So what determines the ordering of the URLs? The matches **must** appear in the vector in decreasing order of score, where the *score* of a URL is the number of occurrences of search terms on its page, counting each search term as many times as it appears on the page. URLs with the same score may appear in the vector in whatever order you want relative to each other.

So here's an example. Imagine we're searching for "milk chocolate zits" and we found two matching pages (at *www.clearskin.com* and *www.acne.com*) that had at least 2 of the 3 terms (since T = 2). Further, let's assume that the word "milk" appears 3 times on the page at *www.clearskin.com* and the word "zits" appears 2 times on that page. The the score for *www.clearskin.com* would be 5. If "chocolate" appears once on *www.acne.com* and "zits" appears twice on that page, then the score for *www.acne.com* would be 3. Since 5 is greater than 3, *www.clearskin.com* must appear closer to the front of the returned vector than *www.acne.com*.

Your *search()* method must run extremely efficiently. Specifically, if there are N words in your index, finding matching pages for a query **must** take far less than O(N) time. If there are M matching pages, producing the returned vector for those pages **must** take far less than $O(M^2)$ time.

Here's an example of how the Searcher class might be used:

```
bool testSearcher()
{
    Searcher s;

      // load a previously-built index
    if ( ! s.load("myIndex"))  // error loading
        return false;

    string query;
    while (getline(cin, query) && !query.empty())
    {
        vector<string> matches = s.search(query);
        if (matches.empty())
            cout << "No pages matched your search terms." << endl;
```

```
        else
        {
            cout << "Your search found " << matches.size()
                 << " matching pages." << endl;
            for (int i = 0; i < matches.size(); i++)
                cout << matches[i] << endl;
        }
    }
    return true;
}
```

Here are the requirements for the Searcher class:

1. It **must** adhere to the specification above.
2. It must **not** use the following STL containers: *C*, unordered_*C*, or hash_*C*, where *C* is map, multimap, set, or multiset. It **may** use other STL containers or your MyMap class if you wish. It **may** use any STL algorithms you wish (hint: *sort()*).
3. It must **not** access any other Impl classes that you write.
4. It **must** be as algorithmically efficient as possible given these constraints.
5. Its member functions must **not** write anything out to cout. They may write to cerr if you like (to help you with debugging).

File I/O

Ok, so how do you read and write to files in C++? And how should you implement the *load()* and *save()* methods of your Indexer class?

The File I/O writeup on the class web site presents the basics of writing to and reading from files, so read that first. This section of the spec details implementation tips for Indexer's *load()* and *save()* methods.

We'd like to be able to read back with *load()* whatever we write with *save()*, whether it be something of a simple type like int or string, or something more complicated, like a list<string> or a MyMap<int,string>. For that reason, we should design a file format for a saved file that is as easy to read as possible.

Here's one approach. We save an item of a simple type by writing it on a line by itself. This is how we would save an int with the value 42 followed by saving a string with the value "gogiberries":

```
42
gogiberries
```

(We can get away with saving one item per line because none of the strings we save in this project will contain newline characters.)

If we has an object of a struct type with two data members, an int and a string, we could save it by writing each data member. If the int data member had the value 42 and the string member had the value "gogiberries", the two lines written would be as above.

To save a container holding a linear sequence of items, such as a list or a vector, we could write each item, one after the other. But if we saved one list<int> followed by another, how would we know where one ended and the other began? A simple solution is to save a sequence container by first writing the number of items, and then writing each item.

What about saving a MyMap? We need to write the key and value at each node of the tree, but in what order should we save the nodes? If we save the nodes in sorted order of their keys, a simple *load()* function that loaded each node and inserted it into the tree would probably build a completely unbalanced tree. We'd like to save the nodes in an order such that inserting those nodes in the same order into a binary search tree would produce a tree with the same structure as the one we saved. Think about why a level-order traversal would do this. (Really, think about this — you'll find the deeper understanding you'll have about trees will be useful.)

Rather than repeatedly writing similar code to save MyMaps with different types of keys and values, the Indexer section above suggested a (non-member) template function; it might be implemented something like this:

```
template<typename KeyType, typename ValueType>
bool saveMyMap(string filename, MyMap<KeyType,ValueType>& m)
{
    Create an ofstream object named, say, stream, to create the file,
        returning false if we can't create the file.
    Save the number of associations in m to stream.
    Visit the associations in m in a level-order traversal of its tree:
    {
        Save the key item of the association to stream.
        Save the value item of the association to stream.
    }
    return true;
}
```

Loading a MyMap is similar:

```
template<typename KeyType, typename ValueType>
bool loadMyMap(string filename, MyMap<KeyType,ValueType>& m)
{
    Clear m of any associations it currently holds.
    Create an ifstream object named, say, stream, to open the file,
        returning false if we can't open the file.
    Read the number of associations in m from stream, returning false
        if we can't.
    As many times as the number read in:
    {
        Read the key item from stream, returning false if we can't.
        Read the value item from stream, returning false if we can't.
        Add the (key, value) association to m.
```

```
    }
    return true;
}
```

Because these are template functions, though, we have a challenge: Reading the value item in *loadMyMap()*, for example, has to use the same syntax no matter what type ValueType is. But we want to read a string using *getline()*, an int using operator>>, a vector using a loop, etc., each of which has a different syntax. How can we do this using the same syntax?

One solution is to bury those details in functions that are all called the same way. We can define a family of (non-member) overloads:

```
bool readItem(istream& stream, string& s);
bool readItem(istream& stream, int& i);
bool readItem(istream& stream, vector<string>& v);
```

Then we implement *Read the value item from* stream*, returning false if we can't* as

```
ValueType value;
if ( ! readItem(stream, value))
    return false;
```

If ValueType is string, this will call the string overload of *readItem()*; if ValueType is vector<string>, this will call the vector<string> overload of *readItem()*. Some *readItem()* implementations themselves might look like this:

```
bool readItem(istream& stream, string& s)
{
      // Call getline and return true for success, false for failure
      // (e.g., end of file: there is no next line in the file to get)
    return getline(stream, s);
}

bool readItem(istream& stream, int& i)
{
    string line;
    if ( ! getline(stream, line))  // return false if getline fails
        return false;

      // create an input stringstream whose text will be a copy of line
    istringstream iss(line);  // #include <sstream> for istringstream

      // Try to extract an int from the stream, returning true if it
      // succeeds, or false if it fails (because the line has no
      // integer).
    return iss >> i;
}
```

A similar technique works for *saveMyMap()*. We might have a family of *writeItem()* overloads; here's one for writing a list of strings:

```
void writeItem(ostream& stream, const list<string>& li)
{
      // write the number of items in the list
    writeItem(stream, li.size());  // calls int overload of writeItem

      // write each item in the list
    for (list<string>::const_iterator p = li.begin();
                                                  p != li.end(); p++)
        writeItem(stream, *p);  // calls string overload of writeItem
}
```

Notice that we delegate the details of writing the size of the list and the strings in the list to the appropriate overloads of *writeItems()*.

So with this scheme, if we had

```
struct Chicken
{
    string name;
    int eggsLaidPerMonth;
};
MyMap<string, list<Chicken> > mslc;
```

then the call `saveMyMap("myChickenMap", mslc)` would compile successfully as long as we have implemented these functions:

```
void writeItem(ostream& stream, string s); // for map key and for Chicken name
void writeItem(ostream& stream, int i);    // for Chicken eggsLaidPerMonth
void writeItem(ostream& stream, const Chicken& c);  // for list elements
void writeItem(ostream& stream, const list<Chicken>& lc); // for map value
```

There's an important consequence of a tricky C++ language rule: The declarations of the *writeItem()* functions (with or without their implementations) must appear **before** you implement the *saveMyMap()* template. The same ordering applies to the *readItem()* overloads and *loadMyMap()*.

Requirements and Other Thoughts

Make sure to read this entire section before beginning your project!

1. In Visual C++, make sure to change your project from UNICODE to Multi Byte Character set, by going to Project → Properties → Configuration Properties → General → Character Set
2. In Visual C++, make sure to add *wininet.lib* to the set of input libraries, by going to Project → Properties → Linker → Input → Additional Dependencies ; otherwise, you'll get a linker error!
3. The entire project can be completed in under 300 lines of C++ code beyond what we've already written for you, so if your program is getting much larger than this, talk to a TA – you're probably doing something wrong.

4. Before you write a line of code for a class, think through what data structures and algorithms you'll need to solve the problem. How will you use these data structures? Plan before you program!
5. Don't make your program overly complex – use the simplest data structures possible that meet the requirements.
6. You must not modify any of the code in the files we provide you that you will not turn in; since you're not turning them in, we will not see those changes. We will incorporate the required files that you turn in into a project with special test versions of the other files.
7. Your Impl classes (e.g., IndexerImpl, SearcherImpl) must **never** ***directly*** use your other Impl classes. They MUST use our provided cover classes instead:

INCORRECT:

```
class WebCrawlerImpl
{
    …
    IndexerImpl m_indexer;  // BAD!
    …
};
```

CORRECT:

```
class WebCrawlerImpl
{
    …
    Indexer m_indexer;  // GOOD!
    …
};
```

8. Make sure to implement and test each class independently of the others that depend on it. Once you get the simplest class coded, get it to compile and test it with a number of different unit tests. Only once you have your first class working should you advance to the next class.
9. You may use only those STL containers (e.g., vector, list) that are not forbidden by this spec. Use the MyMap class if you need a map, for example; do **not** use the STL *map* class.
10. Let *Whatever* represent WordBag, Indexer, WebCrawler, and Searcher. Subject to the constraints we imposed (e.g., no changes to the public interface of the *Whatever* class, no mention of *Whatever*Impl in any file other than *Whatever*.cpp, no use of certain STL containers in your implementation), you're otherwise pretty much free to do whatever you want in *Whatever*.cpp as long as it's related to the support of only the *Whatever* implementation; for example, you may add members (even public ones) to the *Whatever*Impl class (but not the *Whatever* class, of course) and you may add non-member support functions (e.g., saveMap, loadItem, a custom comparison function for *sort()*).

If you don't think you'll be able to finish this project, then take some shortcuts. For example, use the substitute MyMap class we provide instead of creating your own MyMap class if necessary to save time.

You can still get a good amount of partial credit if you implement most of the project. Why? Because if you fail to complete a class (e.g., CrawlerImpl), we will provide a correct version of that class and test it with the rest of your program (by changing our Crawler class to use our version of the class instead of your version). If you implemented the rest of the program properly, it should work perfectly with our version of the CrawlerImpl class and we can give you credit for those parts of the project you completed (This is why we're using Impl classes and non-Impl classes).

But whatever you do, make sure that ALL CODE THAT YOU TURN IN BUILDS without errors with both Visual Studio and either clang++ or g++!

What to Turn In

You should turn in **six** files:

Indexer.cpp	Contains your indexer implementation
Searcher.cpp	Contains your searcher implementation
WebCrawler.cpp	Contains your web crawler implementation
WordBag.cpp	Contains your word bag implementation
MyMap.h	Contains your BST map class template implementation
report.doc, report.docx, or report.txt	Contains your report

You are to define your classes declaration and all member function implementations directly within the specified .h and .cpp files. You may add any #includes or constants you like to these files. You may also add support functions for these classes if you like (e.g., *operator<*, *saveMap*, *loadItem*). Make sure to properly comment your code.

You must submit a brief (You're welcome!) report that presents the big-O for the average case of the following methods. Be sure to make clear the meaning of the variables in your big-O expressions, e.g., "Assuming the Indexer holds W words, and each word is associated with U URLs on average, then *getUrlCounts()* is $O(W^2 \log U)$."

- MyMap: *associate()* and *find()*
- WordBag: constructing a WordBag
- Indexer: *incorporate()* and *getUrlCounts()*
- Searcher: *search()*

Grading

- 95% of your grade will be assigned based on the correctness of your solution
- 5% of your grade will be based on your report

Good luck!