Lab 7: Web Page Indexing

Due: April 9th at 11:59pm

Overview

The goal of this lab is to implement a web page indexing and search program using an AVL balanced binary tree. This lab is paired with Lab 8, and your solution to this lab will be the starting point for that lab. **This is a group lab, groups of two, but you must work with someone in your lab section.**

Deliverables

Your submission should include requisite code necessary to run your program, but the following files will be the focus of grading:

- `Makefile`
- `README`
- `search.cpp`
- `swatHTMLStream.[cpp|h]`
- `test_AVL.cpp`
- `test_SwatHTMLStream.cpp`
- `AVL_tree.h`
- `AVL_tree.inl`
- `AVL_tree_private.inl`
- `AVL_node.inl`
- `library/`
- `inputFiles/`

All starter code is provided via `update35`.

Submission

You will submit using `handin35`. Be sure to place all relevant files in `cs35/labs/07`.

Web Page Indexing

In this lab, you will implement the balancing procedures in an AVL tree and use your tree to complete a web page indexing and keyword searching programming. This is essentially the task of a search engine: Index web pages based on their content and allow users to perform keyword searches over the input. We will constrain the searches to web pages hosted on the `www.cs.swarthmore.edu` domain, and we will provide you a parsing program to read the page content. It is up to you develop an index, which will be a tree-of-trees. The key in the top-level tree is the url being index, and the value is a tree storing the number of occurrences of each word in the web page.

Finally, with your web page index complete, you will allow users to continually search the index. The output of a search will be a list of URLs and the number of occurrences of the keyword in the search. Below is some sample output of a search:

```
bash>./search inputFiles/urls.txt inputFiles/ignore.txt
Loadiing:
        www.cs.swarthmore.edu/~meeden/index.html
        www.cs.swarthmore.edu/~newhall/index.php
        www.cs.swarthmore.edu/~richardw/index.html
        www.cs.swarthmore.edu/~adanner/index.php
        www.cs.swarthmore.edu/~soni/index.php
        www.cs.swarthmore.edu/~aviv/index.html
        (<more URL loads are omitted>)
Enter search query: aviv
Search results for "aviv":
        www.cs.swarthmore.edu/~aviv/index.html:  14
        www.cs.swarthmore.edu/~aviv/classes/s13/cs35/index.html:  1
        www.cs.swarthmore.edu/~aviv/classes/f12/cs43/index.html:  1
        www.cs.swarthmore.edu/~aviv/classes/s13/cs71/index.html:  1

Enter search query: security
Search results for "security":
        www.cs.swarthmore.edu/~aviv/index.html:  17
        www.cs.swarthmore.edu/~aviv/classes/f12/cs43/index.html:  3
        www.cs.swarthmore.edu/~aviv/classes/s13/cs71/index.html:  1

Enter search query: (CTRL^D or EOF to exit)
```

Starter's Abstract As usual, this is an involved lab requiring you to touch many pieces of code, so below is an abstract of the work required. A more complete description of each of the provided files is found at the end of this document.

1. Complete the balancing functions in `AVL_tree_private.inl`
2. Apply the balancing function appropriately in the AVL code, and test your implementation by using the test providing and adding additional test cases to each test function.
3. Work in `search.cpp` to build the webpage index.
4. Complete `search.cpp` so that a user can continually search over the tree and learn which pages contain the keyword.
5. (**Challenge**) Case insensitive searching
6. (**Challenge**) Implement an in-place QuickSort routine to sort the search results.

AVL Implementation and Balance Operations

You should start by completing and testing the AVLTree implementation. The majority of the code to edit is found in `AVL_tree_private.inl`. The AVLTree is very similar to the LinkedBST from Lab 06, except it maintains the AVL balance property to guarantee that the tree is balanced. The AVL balance property is that, for each node in the tree, the height of that node's left and right subtrees differs by at most one.

To efficiently maintain the AVL property, each AVLNode stores the current height of the subtree, as well as a key and value. The class description of an AVLNode can be found below, and the code can be found in `AVL_tree.h`:

```
template <typename K, typename V>
class AVLNode {
  private:
    K key;                      // The key stored in this node.
    V value;                    // The value stored in this node.
    int height;                 // The height of the sub-tree with this node as the root

    AVLNode<K,V>* left;         // Pointer to this node's left subtree.
    AVLNode<K,V>* right;        // Pointer to this node's right subtree.

    AVLNode();                  // Uses the C++ default destructor.

    AVLNode(K k, V v);          // Or can construct using a key and a value

  friend class AVLTree<K,V>;   //AVLTree can read/write private members of AVLNode
};
```

As described in class, an empty tree, e.g. `NULL`, has a height of -1, while a single node with no subtree, e.g., a leaf node, has a height of 0. This is for consistency in calculating the AVL property. You will find a function, `updateHeight()` in the `AVLTree` class that will update the height value of a node based on the heights of the subtree. **Before proceeding with any balance operations, it is always a good idea to ensure that the height value of the nodes involved are accurate.**

To complete the AVLTree, you should complete each of the rotation functions marked `TODO` in `AVLTree_private.inl`. There is one rotation function for each of the four cases discussed in class:

- `balance()` : function for balancing a subtree.
- `rightRotate()` : rebalances the tree when an unbalanced node's left-outer grandchild is too tall.
- `leftRightRotate()`: rebalances the tree when an unbalanced node's left-inner grandchild is too tall.
- `leftRotate()`: rebalances the tree when an unbalanced node's right-outer grandchild is too tall.
- `rightLeftRotate()`: rebalances the tree when an unbalanced node's right-inner grandchild is too tall.
- `updateHeight()` : updates the height of current node

The rotation functions will be called by `balance()`, which should be calledwithin the `insertFromSubtree()` and `removeFromSubtree()` private recursive functions to maintain the AVL property when tree heights may change. `balance()` will not only rebalance the tree, but it will also detect an imbalance, which is why you need to keep the heights updated. Finally, `balance()`, like some of the other recursive functions, will return the root of the balanced subtree, which may be the incoming root if the subtree did not need balancing.

Testing your AVL

For this lab, a simple test program is provided. **You are required to add at least two additional assert tests to each of the test functions** to test your tree. One assert test should check that removes balancing is working, and the other should test that additional inserts are working, perhaps on larger trees. You may draw on code from previous labs, such as `test_BST.cpp`, to help your tests.

Indexing and Searching Web Pages

Your main work for this lab is to write a search program, inside `search.cpp` that analyzes `www.cs.swarthmore.edu` web pages and allows a user to find pages that match a search query. This program should take two command-line arguments:

- urls-file: a file of www.cs.swarthmore.edu URLs (whitespace-separated) to analyze.
- ignores-file: a file of search terms to ignore (also whitespace-separated)

And, your program should do the following:

- Open and read the ignores-file, saving each word in the file into a tree containing all words to ignore when searching web content.
- Open and read the urls-file. For each URL in the file, use the SwatHTMLStream to read the web page for that URL, creating an index of the word frequencies for that web page. In your indexes do not include any words contained in the ignores-file.
- Using standard input and output, allow the user to search for terms in the web pages. For each search term, display the URL of each page containing the term and the number of occurrences of the term in that web page.

Reading data from files

To open a file for reading in C++, you need to create a file `iostream`, more generally referred to as a `fstream`. File streams are declared on the stack, an example is below using an *input* file stream, or an `ifstream`.

```
//...
#include <fstream>
//...

string input_file = "YOUR_INPUT_FILE";
ifstream myFile(input_file.c_str()); //require a c string, not c++
if (!myFile.good()) {
  //ERROR condition!
}else{
  string line;
  myFile >> line; //read a line of the file
  while (!myFile.eof()) {
     cout << line << endl; //print the line
     myFile >> line; //read next line
   }
   myFile.close(); //close the file
}
```

You should refer to the documentation online for more details on file streams, found here `http://www.cplusplus.com/doc/tutorial/files/`. This is very useful stuff.

SwatHTMLStream

For this lab we have also provided SwatHTMLStream, a stream-like class that allows you to read HTML files hosted by the local CS web server, `www.cs.swarthmore.edu`. To use the SwatHTMLStream you must be logged into a Swarthmore Computer Science lab computer. You can use the SwatHTMLStream much like a file stream:

```
string URLName = "www.cs.swathmore.edu/~aviv/index.html"; //url to retrieve
SwatHTMLStream file(URLName.c_str()); //require a c string, not a c++ string

if(file.good(){
  string word;
  file >> word;//read one word from the file
  while(! file.eof()){
    //do something with the tree
     file >> word; //read next word
  }
  file.close();
}
```

The SwatHTMLStream is not a fully-featured input stream, but can read strings of words (and number-strings) from a given URL. It skips HTML tags (for correctly-formatted HTML) and most punctuation characters, and converts all strings to lowercase.

Handling Error

Some of the URL lists contain URLs that cannot be retrieved by the SwatHTMLStream. **This will cause error return values, and you should handle these errors appropriately**. Your search program should not stop on reading a bad URL, and it s should move onto the next URL in the list when an error occurs. However, early in your development, you may want to leave in stricter checks to make sure key functionality is sound.

URL Files

We have provided you with a number of different URL files:

- `all_CS.txt` : This contains every html/php page hosted under the CS domain. Test with this file last.
- `all_CS_html.txt` : This contains all the html pages hosted under the CS domain..
- `all_CS_php.txt` : This contains all the php pages hosted under the CS domain.
- `urls-short.txt` : This contains a short set of URLs that is good for initial testing.
- `urls.txt` : This contains a longer set of URLs that is good for secondary testing.

Command-line arguments in C/C++

To process command-line arguments in C++ you need to add two arguments to your main function:

```
int main(int argc, char** argv) {
   // ...
   return 0;
}
```

`argc` is an integer representing the number of arguments entered (including the program name itself, that is, `argv[0]` is the program name), and `argv` is an array of character arrays, with each character array containing one argument to the program. (These variables can be called anything you want, but it's a long-standing convention to call them `argc` and `argv`.) You can then use `argc` and `argv` inside your program to access the command-line arguments. For example, if you run your program:

```
> ./search inputFiles/urls.txt inputFiles/ignore.txt
```

`argc` has the value 3 (there are three parameters). That means that `argv` is a c-style array with three values. `argv[0]` is the value "search", `argv[1]` is "inputFiles/urls.txt" and `argv[2]` is "inputFiles/ignore.txt". `argc` is useful to check if the user entered the correct number of arguments. For example, if the user forgets the arguments you can print an error:

```
$ ./search
Usage: search <url-file> <ignore-file>
Required options
  <url-file>:     a file of urls to index and search
  <ignore-file>:  a file of words to ignore
```

Creating your own webpage

If you want to create your own web page as a test (or otherwise), do the following:

1. In your home directory, make a `public_html` directory to hold the web page, then make that directory readable and executable for everyone:

   ```
   > mkdir ~/public_html
   > chmod a+rx ~/public_html
   ```

2. Inside your `public_html` directory make an `index.html` file, then make that file readable by everyone:

   ```
   > chmod a+r ~/public_html/index.html
   ```

3. You can start with a simple text file (e.g., Hello world!) and add more to it later. (See other users' web pages for examples of what you can do in HTML.) After you've saved your index.html file and made it world-readable, use a web browser to view:

   ```
   http://www.cs.swarthmore.edu/~yourUserName
   ```

Grading and Extra Credit

Rubric

This lab will be graded on a 50 point scale:

- (15 Points) Balance Functions properly implemented and applied to the BST code.
- (5 Points) Additional Unit-Tests
- (20 Points) Web page index search and traversal
- (10 Points) Coding Style and memory

Note that there are points dedicated to aspects of your lab that are not part of the solution. Do pay attention to coding style and managing memory, design and lay out your code effectively so that it is readable and easily debugged. Run your program through `valgrind` and plug any memory leaks.

Challenge and Extra Credit

There are challenge exercise on this lab. The next lab in the sequence has a significant extra credit component.

- **(Challenge)** : Add case insensitive searching and indexing.
- **(Challenge)** : Implement an in-place QuickSort routine that will sort the search results based on the number occurrences of a word in the webpage.

Provided Code

- `Makefile` : The make file, you will not need to edit this file unless you add additional test
- `search.cpp` : The main part of the program. You will need to edit this file.
- `swatHTMLStream.[cpp|h]` : The html parsing and reading files. You do not need to edi these files, but you should read them.
- `test_AVL.cpp` : The test file for the AVL tree. You will need to edit this file.
- `test_SwatHTMLStream.cpp` : The test file for the SwatHTMLStream. You will not need to edit this file/
- `AVL_tree.h` : The header file for the AVL tree. You will not need to edit this file.
- `AVL_tree.inl` : The inline implementation of the public facing functions of the AVL tree
- `AVL_tree_private.inl` : The inline implementation of the private functions. You will need to edit this file.
- `AVL_node.inl` : The inline implementation of teh AVL node. You will not need to edit this file.
- `library/` : The library contains a Queue and Circular Array List implementatoin, as well as BST and List header files.
- `inputFiles/` : The directory containing a bunch of URL lists, and a sample ignore list.

Compilation

Compilation will occur using `make`. To compile the amazing

```
bash> make
```

If you add extra functionality contained in other files, you should edit the `Makefile`, but it is not necessary. To compile the test code:

```
bash> make test
```

Which will compile an executable `test_BST` at first, you can add more tests as you develop.

Execution

Here are the command line arguments for `crack`:

```
>./search
Usage: ./search <url-file> <ignore-file>
Required options:
  <url-file>:     a file of urls to index and search
  <ignore-file>:  a file of words to ignore
```