COMS 3136 Essential Data Structures in C/C++, Lab #4 (200 points total)

Please read this assignment carefully and follow the instructions EXACTLY.

Submission

Please refer to the lab retrieval and submission instruction, which outlines the only way to submit your lab assignments. Please do not email me your code.

If a lab assignment consists of multiple parts, your code for each part must be in a subdirectory (named "part1", "part2", etc.)

Please make sure that your submission satisfies the requirements for the following items:

- README.txt
- Makefile
- Valgrind

The requirements remain the same as lab 1.

Part 1: ncat in C++ (40 points)

Recall ncat.c from the lecture note on I/O:

```
/*
 * ncat <file_name>
 *
 *   - reads a file line-by-line, printing them out with line numbers
 */

#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv)
{
    if (argc != 2) {
        fprintf(stderr, "%s\n", "usage: ncat <file_name>");
        exit(1);
    }

    char *filename = argv[1];
    FILE *fp = fopen(filename, "r");
    if (fp == NULL) {
        perror(filename);
        exit(1);
    }

    char buf[100];
    int lineno = 1;
    while (fgets(buf, sizeof(buf), fp) != NULL) {
```

```
            printf("[%4d]", lineno++);
            if (fputs(buf, stdout) == EOF) {
                perror("can't write to stdout");
                exit(1);
            }
        }

        if (ferror(fp)) {
            perror(filename);
            exit(1);
        }

        fclose(fp);
        return 0;
    }
```

You are given ncat-main.cpp, a C++ re-implementation of the ncat program:

```
[   1]// DO NOT MODIFY THIS FILE.
[   2]
[   3]#include <cstdlib>
[   4]#include <iostream>
[   5]#include <iomanip>
[   6]#include <string>
[   7]using namespace std;
[   8]
[   9]#include "myfilereader.h"
[  10]
[  11]int main(int argc, char **argv)
[  12]{
[  13]    if (argc != 2) {
[  14]        cerr << "usage: ncat <file_name>" << endl;
[  15]        exit(1);
[  16]    }
[  17]
[  18]    const char *filename = argv[1];
[  19]    MyFileReader f(filename);
[  20]
[  21]    string line("");
[  22]    char buf[10];
[  23]    while (1) {
[  24]
[  25]        // MyFileReader::getLine(char *buffer, int size) reads
[  26]        // at most (size - 1) characters from the file and
[  27]        // stores them in the given buffer.
[  28]        // Reading stops when a '\n' is read,
[  29]        // or when the end-of-file (EOF) is reached.
[  30]        // The newline, if any, is retained.
[  31]        // If any characters are read, the buffer is null-terminated.
[  32]
[  33]        int result = f.getLine(buf, sizeof(buf));
[  34]
[  35]        if (result == -1) {
[  36]            // getLine() returns -1 if EOF has occurred before
[  37]            // any characters are read.
[  38]            //
[  39]            // In this case, we print the partial line that we have
[  40]            // collected so far, if any, and get out of the loop.
```

```
[  41]            if (line.length() > 0) {
[  42]                int lineno = f.getCurrentLineNumber();
[  43]                cout << "[" << setw(4) << lineno << "]";
[  44]                cout << line << flush;
[  45]            }
[  46]            break;
[  47]        }
[  48]        else if (result == 0) {
[  49]            // getLine() returns 0 if some characters are read into
[  50]            // the buffer, and the last one read is NOT a '\n'.
[  51]            //
[  52]            // In this case, we simply keep collecting them.
[  53]            line += buf;
[  54]        }
[  55]        else if (result == 1) {
[  56]            // getLine() returns 1 if some characters are read into
[  57]            // the buffer, and the last one read is a '\n'.
[  58]            //
[  59]            // At this point, an entire line has been read.
[  60]            // We print it, and start collecting the next line.
[  61]            int lineno = f.getCurrentLineNumber() - 1;
[  62]            cout << "[" << setw(4) << lineno << "]";
[  63]            cout << line << buf << flush;
[  64]            line = "";
[  65]        }
[  66]        else {
[  67]            cerr << "ERR: getLine() returned unknown result." << endl;
[  68]            exit(1);
[  69]        }
[  70]    }
[  71]    return 0;
[  72]}
```

Note that the ncat-main.cpp code with line numbers shown above is in fact the output of the program when we ran it on the ncat-main.cpp file.

The C++ version fixes the deficiency of the C version, which incorrectly inserts line numbers in the middle for any long lines that do not fit in the fixed-size buffer.

The C++ program uses MyFileReader class. Your job is to implement MyFileReader class in myfilereader.h & myfilereader.cpp files.

Some requirements and tips:

- Don't forget your Makefile. Call your executable "ncat".
- Do NOT modify ncat-main.cpp. It will be replaced with the original version for grading.
- If you encounter any error within the member functions of MyFileReader class, print some helpful error message and exit the program in the same manner as the C version of the program.
- As you can see in the listing above, the line numbers starts with 1.
- Valgrind-clean execution will be a big part of your grade. No Valgrind error of any kind is allowed.

Part 2: Tokenizing input (40 points)

Add the following two member functions to MyFileReader class:

```
void MyFileReader::tokenizeLine(vector<string>& vec);
bool MyFileReader::haveAllLinesBeenRead() const;
```

You can start by coping over the myfilereader.h/cpp files from part1 directory, and add the member functions.

Here is what tokenizeLine() function does:

- It reads an entire line from the file (perhaps utilizing the getLine() function you wrote in part1).

- It picks out the words in the line. Words are separated by white-spaces. White-space includes space, tab, and newline.

- For each word, it removes all characters that are not English alphabets, and then lowercases the word. If the result is a non-empty string, it adds the string to the vector.

Some tips and requirements:

- The easiest way to split a line on white-space is to use istringstream class in the C++ Standard Library. Here is a sample code that shows you how to use it:

```
[   1]#include <iostream>
[   2]#include <sstream>
[   3]using namespace std;
[   4]
[   5]int main()
[   6]{
[   7]    string str("   SOME  LONG    STRING\twith\nSPACES    ");
[   8]
[   9]    istringstream iss(str);
[  10]
[  11]    string s;
[  12]    while (iss >> s) {
[  13]        cout << "[" << s << "]" << endl;
[  14]    }
[  15]    return 0;
[  16]}
```

- You may find the following functions useful as well:

```
string::operator+=(char);
isalpha(char);
tolower(char);
```

- Note that the tokenizeLine() function takes a vector by reference, enabling the function to modify the original vector passed in. Here is how you would construct an empty vector of string and pass it to tokenizeLine():

```
        vector<string> v;
        f.tokenizeLine(v);
```

  - The haveAllLinesBeenRead() member function returns true if the EOF has occurred on the file.

Write the tokenize program (code in tokenize.cpp & executable named "tokenize") that reads a file, prints tokens for each line along with the line number (only if there is at least one token), and finally prints out the total number of tokens extracted.

Here is the result of running tokenize on the istringstream sample program:

```
    $ ./tokenize istringstream-test.cpp
    [   1]{ 2 tokens} include iostream
    [   2]{ 2 tokens} include sstream
    [   3]{ 3 tokens} using namespace std
    [   5]{ 2 tokens} int main
    [   7]{ 5 tokens} string str some long stringtwithnspaces
    [   9]{ 2 tokens} istringstream issstr
    [  11]{ 2 tokens} string s
    [  12]{ 3 tokens} while iss s
    [  13]{ 3 tokens} cout s endl
    [  15]{ 1 tokens} return
    25 tokens total
```

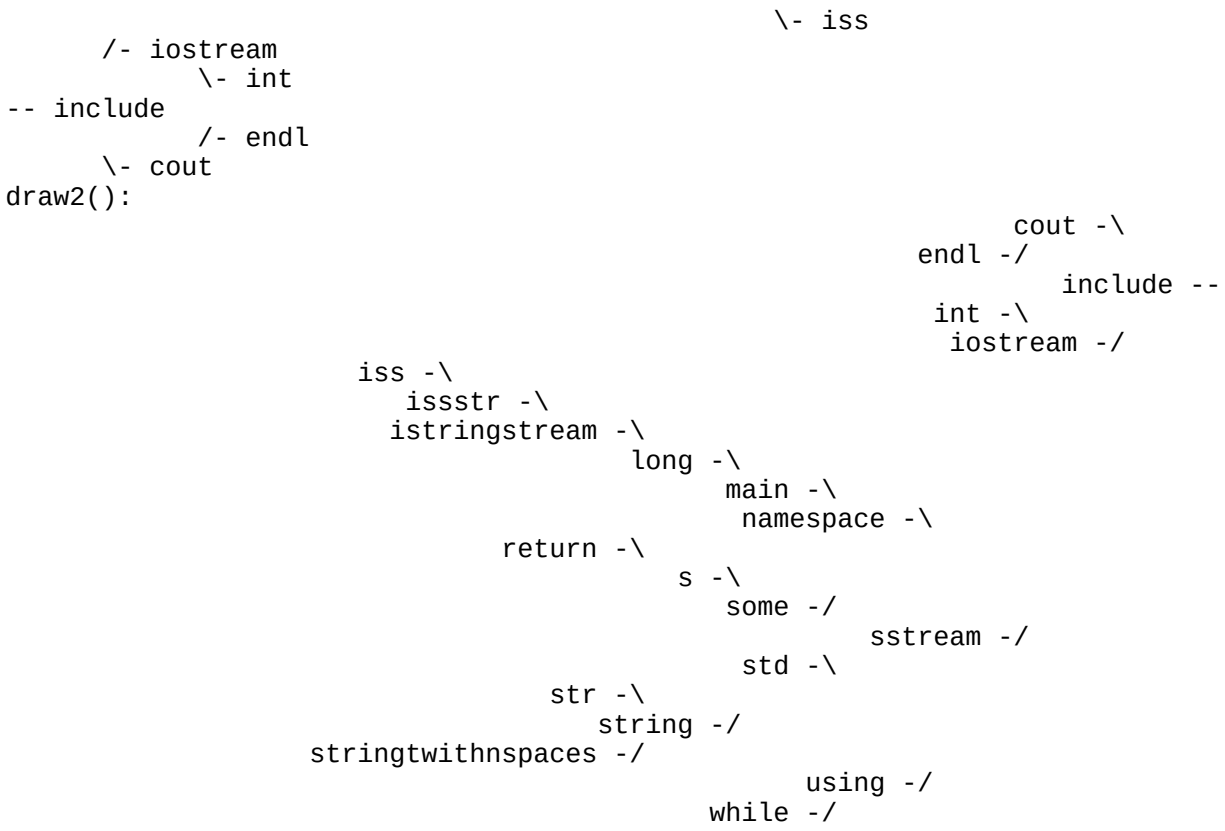
Part 3: BST of tokens (40 points)

Write a program that tokenizes an input file like the tokenize program in part2, but builds a binary search tree (BST) of the tokens.

Your program then draws the BST twice: the first one is drawn rotated 90-degree counter-clockwise, using the BST::draw() function we learned in class, and the second one is drawn rotated 90-degree clockwise, using BST::draw2() function you will write in this part.

Here is the sample run:

```
$ ./tree-test ../part2/istringstream-test.cpp
draw():
                        /- while
                  /- using
                                 /- stringtwithnspaces
                           /- string
                                 \- str
                        \- std
            /- sstream
                        /- some
                              \- s
                                 \- return
                  \- namespace
                        \- main
                              \- long
                                 \- istringstream
                                    \- issstr
```



Some requirements and tips:

- Don’t forget your Makefile and name your executable "tree-test".

- You can take BST-2.h/cpp and turn it into a string version.

- Or if you feel ambitious, you can make BST a template class. Turning a non-template code into a template is conceptually straight-forward and fairly a mechanical process, but can still present a challenge because it is easy to make a syntactical mistake, and the compiler often throws completely cryptic error message.

  You can ask for help in the mailing list. You are allowed to post code as long as it is about translating the existing BST-2 code into a template syntax.

- The output of draw2() must preserve the formatting shown in the sample output above. Specifically, the edges at the same depth must print at the same column of the screen.

  Also note that the strings are printed right-justified. You may assume that the maximum length of any token is 25. That assumption allows you to right-justify the strings using the same technique I used for line numbers in part1.

Part 4: Spell checker (40 points)

In part4, you will write a spell checking program. Here is a sample run:

```
$ ./spell-check /home/jae/cs3136-pub/data/words istringstream-test.cpp
Dictionary info:
  Tokens read:   98568
  Tree size:     85896
  Tree height:   1885
  Balance index: 110.941

Misspelled tokens in istringstream-test.cpp:
[   1] iostream
[   2] sstream
[   3] namespace std
[   5] int
[   7] str stringtwithnspaces
[   9] istringstream issstr
[  12] iss
[  13] cout endl
```

Requirements and tips:

- Name your executable "spell-check".

- First, the program reads a dictionary file (given as the 1st command line argument), tokenizes the words in the same way we did in part 2 & 3, and builds a dictionary BST. Note that, because of the tokenizing process (i.e. removing non-alphabets and lowercasing), the BST may end up containing fewer number of words than what was read from the dictionary file. In the sample run above, the file contained 98568 words, but the BST has 85896 nodes.

- There is a large dictionary file: /home/jae/cs3136-pub/data/words

- BST should be built by inserting each token in the order it appears in the dictionary file.

- The program prints the height of the dictionary BST.

- The program also prints the "balance index", which is defined by the following formula:

  ```
  (tree_height + 1) / ceil(log2(tree_size + 1))
  ```

  It is a measure of how balanced a binary tree is. A perfectly balanced tree will have the minimum possible value, 1.0.

- Then, the program reads an input file (given as the 2nd command line argument), tokenizes the words in the same way, and spell-check each token. A token is considered misspelled if it’s not in the dictionary. For each line of the input file that contains one or more misspelled tokens, the program prints the line number and misspelled tokens, as shown in the sample run above.

Part 5: Improving performance (40 points)

In this part, you will make an improvement to the spell-check program for better performance. Specifically, your goal is to implement a solution that will reduce spell-check's balance index to 5.0 or lower on any sufficiently large dictionary file (such as the one used in part4).

Some requirements and tips:

- If your solution involves randomness, you should achieve 5.0 or lower with high probability.

- You are free to implement whatever solution you can come up with, as long as it works correctly and achieves < 5.0 balance index. Be creative!

- You can even search on the Internet to get ideas and to learn relevant algorithms. But your code must be of your own. I know what's out there. For suspected plagiarism, I reserve the right to call you in and grill you in person on the finer points of your code.

In addition to implementing the solution, answer the following questions in your README.txt:

(a) First, describe the performance problem of the part4 version. Why does it have such a high balance index?

(b) What is the maximum value of balance index that a binary tree can have? Write the formula in C language (i.e. the way I wrote the formula for the balance index).

(c) Describe your solution. How does it work? What balance index does it achieve? How did you come up with the solution? Cite any source that you have consulted.

(d) Measure and compare the runtime of the two versions of spell-check. Use the dictionary file I provided. For input, use 3 different files with different sizes: small, medium, large. (You determine what small, medium, large mean.) Analyze the result.

(e) Measure and compare the memory usage of the two versions of spell-check. Does you solution require more memory? If so, by how much? Note that there is a bug in the GNU 'time' command version 1.7: it reports 4 times the actual usage. So you need to divide the reported number by 4.

--

Make sure that all your executables run with no valgrind error.

Good luck!