Assignment 01 (60 points)

Due: 11:59pm, January 26th

You must pass 100% of the auto-grading on gradescope to be able to enroll or stay enrolled in this class. Note the **autograder** on gradescope **will be available after the first class**.

You must do this assignment **on your own**. A single programmer affidavit of academic honesty will need to be submitted along with your code.

Start on this assignment and ask questions early**. Do NOT expect help** from TAs or instructor **on the due date.**

Why this prerequisite assignment?

Most modern operating systems (UNIX™ based OS, Windows, etc.) were written in C/C++ and assembly languages. For the purpose of learning in this class, you are required to do all programming assignments in C++ or C language. This first assignment is designed for you to demonstrate necessary knowledge and proficiency in C++ or C programming as part of the prerequisites to enroll in this class.

This assignment is applicable to the currently enrolled students as well as the waitlisted students. We need to give all waitlisted students a fair chance to enroll.

Note for **waitlisted** students who choose to do this assignment and pass the auto-grading, you will be added to the class **only if** there are spots opened after the prerequisite assignment. Please also note it is a mandate from the university and the CS department to keep class size to be within a manageable size to make sure we can support all students.

We understand that many of you are looking for graduation this coming May. However, we cannot relax this prerequisite requirement, as again, we need to be fair to every student trying to enroll.

NO Late Submission

As this assignment is used for determining your enrollment eligibility, for the fairness and uniformity of grading, **NO LATE submission** will be accepted for this assignment, we also need time to sort out the grading before Jan 28 which is the deadline for instructor to drop students. There will be **NO extension** to this submission. All students (currently enrolled and waitlisted) must submit this assignment by due time if you intend to remain in the class or enroll.

Grading

Passing 100% autograding will clear the programming prerequisite to enroll in the class, but it may NOT give you a perfect score on the assignment. The structure, coding style, and commenting will also be part of the rubrics for the final grade (**see Syllabus Course Design - assignments**). Your code shall follow industry best practices:

- Be sure to comment your code appropriately. Code with no or minimal comments are automatically lowered one grade category.
- Design and implement clean interfaces between modules.
- NO global variables.
- NO hard code – Magic numbers, etc.
- Have proper code structure between .h and .c / .cpp files, do not #include .cpp files.

All test data including one correct program output are given to you, **any hardcoding** to generate the correct output without implementing the dictionary tree will automatically **result in a zero grade** of the assignment.

Academic honesty

Posting this assignment to any online learning platform and asking for help is considered academic dishonesty and will be reported.

An automated program structure comparison algorithm will be used to detect code plagiarism.

- The plagiarism detection generates similarity reports of your code with your peers as well as from online sources. We will also include solutions from the popular learning platforms (such as Chegg, etc.) as part of the online sources used for the plagiarism similarity detection. Note not only the plagiarism detection checks for matching content, but also it checks the structure of your code.
- If plagiarism is found in your code, you will be automatically disenrolled from the class. You may also get reported.
- Note the provided source code snippets for illustrating proposed implementation would be ignored in the plagiarism check.

Task

Tree, a hierarchical data structure, is often used in representing OS entities (such as the organization of a file system, Unix-style process trees, etc.), and in implementing various OS algorithms (such as multi-level paging table for memory management, multi-level indexed files, etc.).

In this assignment, you will use C or C++ to implement a dictionary tree, and write test code for searching and counting words in the dictionary tree:

1) You must implement the **pointer tree data structure** (**see dictionaryTreeSpec.pdf**).
2) Tree operations:
    a. Insert a word to the tree.
    b. Count all words in the tree that start with a specific string.
3) Test code (see **program execution and required output** below):
    a. Read words from a dictionary text file and insert them to a dictionary tree.
    b. Read words from another text file, and **for each word read in**: search, count, and print the number of words in the dictionary tree that start with it.

Requirements for Compilation and Execution:

1) You must create and submit a Makefile for compiling your source code. Makefiles are program compilation specifications. See below (also refer to the Programming page in

Canvas) for details on how to create a makefile. A makefile example is also provided to you.

2) The make file must create the executable with a name as **countwords.**
3) Your code **must compile and run correctly on edoras.sdsu.edu**, regardless of where you wrote, compiled, and tested your program. Excuses of "but it worked on my machine" will **NOT** be accepted.
    a. The compiler used for autograder is gcc version 4.8.x, as provided in Edoras. To use C++ 2011, you need to use the flag -std=c++11.
    b. Refer to the Programming page in Canvas for C/C++ programming and testing under Linux environments.
    c. If you are **already enrolled** in the class, your Edoras account information will be emailed to you. If you didn't get an email for your Edoras account by the first class, please email your TAs asap for it.
    d. If you are a **waitlisted** student, please **email TAs** to get an Edoras account asap.
4) The executable **requires two command line arguments**:
    a. The **first argument** specifies the file path to the source vocabulary text file that provides the words for building the dictionary tree.
        i. Use **dictionarysource.txt** file as the source vocabulary text file for testing.
    b. The **second argument** specifies the file path to the text file that provides the words to be searched in the dictionary tree.
        i. Two files **testfile1.txt** and **testfile2.txt** are given for your testing.
        ii. Auto-grading on gradescope will use all or **part** of **testfile1.txt** and **testfile2.txt**.
            - Test case 1 uses testfile1.txt, **testfile1SolutionOutput.txt has the correct output for execution using testfile1.txt (see below)**
            - Test case 2, 3 and 4 use different portions of testfile2.txt
    c. **Your program should have minimal error checking** on whether the command line arguments are provided from execution, fail gracefully when there is a wrong number of arguments, or a file does not exist.
5) Program execution and **required output:**
    a. Using the following command line **as an example**:
        i. ./countwords dictionarysource.txt testfile1.txt
    b. Your program would first read the words from the **dictionarysource.txt** and insert them to a dictionary tree.
    c. It would then read the words from **testfile1.txt**, in the **order** of how they appear in the file, and **for every word read in from testfile1.txt**:
        i. **search and count the number of words** in the dictionary tree that **start** with this word,
        ii. and **print this word and the count from 5) c. i. above (separated by a space) to the standard output ONE line per word**, in the exact format as:
        hour 10
        'hour 10' here means for the 'hour' read from testfile1.txt, there are 10 words in the dictionary tree that start with 'hour'.
        iii. More examples:

- If the dictionary tree has bat, batch, batman words inserted; there should be three words starting from 'bat', your program shall print:

  bat 3

- If the dictionary tree has education, educate, edition words inserted; there should be three words starting from 'ed', but there should be 0 words starting from 'editor', your program shall print:

  ed 3
  editor 0

iv. Note the same word can appear multiple times from the test file. For example, the word 'hour' appears many times in test1.txt, your program would need to do the i. and ii. above every time it reads in an 'hour' word.

d. See **testfile1SolutionOutput.txt** for the correct output from this execution. Using the same command line arguments, your program should print the output **to the standard output** exactly as what is contained in **testfile1SolutionOutput.txt.**

e. **REMEMBER**: **Do NOT** print the output to a file, print to the **standard output; otherwise, autograding will fail.**

Turning In to Gradescope

Make sure that all files mentioned below (Source code files and Makefile) contain your name and Red ID! **Do NOT compress / zip** files into a ZIP file and submit, submit all files separately.

- Source code files (.h, .hpp, .cpp, .c, .C or .cc files, etc.)
- Makefile
- A sample output (in a text file) from a test of your program. (use > to export standard output to a file)
- Single Programmer Affidavit with your name and RED ID as signature, use Single Programmer Affidavit.docx from Canvas Module Course Resources – Programming

If you are a waitlisted student, please enroll yourself asap to the gradescope by using the entry code that is emailed to you from Naedean.

Makefile

a. A Makefile is for compiling and linking C/C++ code to generate an executable file. The sample Makefile provided is for compiling and linking C++. Suppose you have dictionary.h and dictionary.cpp for dictionary tree implementation, and countwords.cpp having the main function that drives the dictionary population and word counting.
   - CXX is the variable specifying the C++ compiler as g++ (note autograder uses g++ compiler version 4.8.x, the same as what's installed on Edoras)
   - CXXFLAGS (-std=c++11 -g) specifies compilation flags to the compiler specified by CXX, instructs the compiler to use the ISO 9899 standard C++

implementation published in 2011 (commonly called c++11, c11 for C), c++11 and c11 have functionality that is desirable (e.g., the bool type in C and the type safe nullptr in C++). -g adds debugging information to the executables. Debugging information lets the use of the GNU symbolic debugger (gdb) to debug your programs.

If you are writing in C, you would change the CXX= and CXXFLAGS= to:

- CC=gcc
- CCFLAGS=-std=c11 -g

b. To compile your code, simply type **make** at the prompt. For the C++ version, make will execute compilation similar to the following if you do not have any errors:

```
g++ -std=c++11 -g   -c -o dictionary.o dictionary.cpp
g++ -std=c++11 -g   -c -o countwords.o countwords.cpp
g++ -std=c++11 -g -o dictionary dictionary.o countwords.o
```

With either C or C++, -o specifies the output file. The -c flag implies that we are compiling part of a program to an object file (machine code, but not a stand-alone program). Makefiles usually do this as it permits us to not recompile the whole program when only one module has changed. The last line links the object files together and adds some glue to create an executable program.

Programming environments

There are a number of graphical user interface front ends for coding and debugging, ranging from simple interfaces in emacs and vim to eclipse and Microsoft's cross-platform Visual Studio Code. For more on this, please refer to the Programming page under Canvas module – Course Resources - Programming.