Assignment 01 (70 points)
Due: Beginning of class, **01/30/2024**
Late Due: Beginning of class, **02/01/2024**
You must work on your own.

The Unix™ operating system is almost entirely written in C programming language, as is true for most modern operating systems including Windows, etc. As a major part of your learning activities for this course, you are required to use C or C++ for doing the programming assignments. It is assumed that you had some form of training in C or C++ programming prior to the class. If that was not the case, I would recommend you to seriously reconsider your enrollment.

This first assignment is designed for you to reacquaint yourself with C/C++ programming as well as the tree data structure, which is often used in representing OS entities, e.g., the organization of a file system, Unix process tree, multi-level paging table for memory management, multi-level indexed file allocation table, etc.

Task

Suppose your company is establishing a new engineering organization, and you are asked to implement a tree data structure to store and retrieve the organization chart information, as well as write code for testing your implementation.

1) Tree operations:
    a. Add employees identified along an employee ID path to the organization chart, see below for elaboration and examples.
    b. Find an employee in the organization chart based on an employee ID path.
    c. Count number of employees in the organization chart starting from an employee ID path.
2) Test code (see program execution and **required output** below):
    a. Read an org chart file that has the information of levels and employee ID paths to build an org chart.
    b. Read another file containing employee ID paths, and for each employee ID path read, count, and print the number of employees starting from it in the org chart.

Tree specifications

Node representation

You may use either a C struct type or a C++ class to define the employee node. The **key member** of a tree **node representation is the children node array**. For practicing the pointer use in this assignment:

- You **must** use **a double pointer** for specifying the children node array.
    - e.g., **EmployeeNode ** children;**

    - Do **NOT** use any **collection** data structure from the STL (standard template library), such as vector, list, etc.
- **Violating the above requirement would result in a zero point** for a1 assignment.
- You may add other data members to your node structure as you see needed, such as a member for employee ID as a string (use any available string type from libraries for what you like), node level, etc.
- Programming tips:
    - Sample code for instantiating a double pointer array,
        - children = new EmployeeNode*[numOfChildren]; // C++
        - children = (struct EmployeeNode**) malloc(numOfChildren*sizeof(struct EmployeeNode*)); // C
    - As a best practice, after instantiation, always **explicitly initialize** all child pointers to NULL or nullptr. And in C and C++, it is always a best practice to explicitly initialize all variables before using them.
        - The "run in one environment but not in another" problem is often caused by non or improper initializations.

Tree operations

Implement the following tree operations with the specified logic (Return type and function name can vary, but pay attention to the **requirements** below):

- First, using **iterative or recursive** implementation is your choice.
- **bool addEmployees(const char *employeeIDPath);**
    - You are **required** to use const **char*** for the employee ID path argument, you need to use C string manipulation to parse and extract information from the employeeIDPath.
        - You must **not** use std::string for parsing this argument in the function.
        - **Violating this requirement would result in a 50% deduction** of your a1 grade.
        - You may use std::string in other contexts of your program, e.g., at the end of the Appendix (see below), std::string is used for reading in lines from a text file.
    - Note this signature is for a member function in a C++ node class; if you are using C, you would need to add an argument for specifying which node to start with, e.g.:
        - bool addEmployees(struct EmployeeNode *curNode, const char * employeeIDPath);
        - **Similarly,** this applies to other functions below.
    - **Logic**:
        - Starting from the current node, **insert** employee nodes identified along the employee ID Path to the tree, **stop** if the employeeIDPath cannot be parsed further due to **bad format** or **reaching the end of the path** or **reaching** beyond the **total number of levels** specified from the file for building the org chart (see below the orgchart.txt file).
        - Inserting an employee node at a child location of a node is to create a new employee node and assign the newly created node to the location.

        - Each employee node should also have an **employee ID** (as a string) set to it, see the sample invocation below on what the employee ID should be set.
    - The **employee ID** path argument should contain a series of **single digit** integers separated by an underscore, e.g., 0_2_3, with each number representing the child index at the appropriate level starting from the current node, and they are **zero based**.
        - Each index must be within the max number of children allowed for the level minus 1, as indices are zero based. And **assume** each index is within a single digit range [0 – 9]
            - The max allowed number of children for each level will be from the file specified in the first command line argument (see below orgchart.txt in compilation and execution section)
        - If an index is outside the allowed range or is not a digit, **stop** the adding process and return.
            - Note the added nodes so far will stay.
    - **Sample invocation** with employee ID path as 0_2_3:
        - The digits in the path are zero based indices of the child array for the levels.
        - Starting from current node, **add or insert** an employee (**if it is NOT added yet**) as 0th child of the current node and set its employee ID string to e_0, then add an employee as the 2nd child under employee e_0 and set its employee ID to e_0_2, then add an employee as the 3rd direct report under employee e_0_2 and set its employee ID to e_0_2_3
        - **Note**: the **head** (root) node of the org chart should always have an employee ID as **e**
        - Always check if the child index is within the child index range for the level before creating and adding the child employee node.
            - See above.
        - Always check if the number levels that is traversed has exceeded the total number of levels from the org chart file.
        - When called on the root / head node, employeeIDPath should contain the full employee ID path to be added.
- **EmployeeNode* findEmployee(const char *employeeIDPath);**
    - **Logic:**
        - find and return the employee node in the org chart based on an employee ID path (e.g., 0_2_3)
        - See above for the elaboration on the employee ID path.
    - Same requirement applies as the addEmployees function above:
        - You are **required** to use const **char*** for the employee ID path argument, you need to use C string manipulation to parse and extract information from the employeeIDPath.
            - You must **not** use std::string for parsing this argument in the function.
            - **Violating this requirement would result in a 50% deduction** of your a1 grade.
- **void countNumOfEmployeesInOrgchart (unsigned int &count);**

- You may change the signature of this function (function name or return type or argument) for what you like as long as the logic is accomplished as specified below
- **Logic:**
  - count the number of employees under the org chart starting from the current node.

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 **countEmployees.**
3) The executable **requires two command line arguments**:
   a. The **first argument** specifies the file path to an **orgchart** text file that provides the information for building an org chart tree:
      i. An **orgchart.txt** file is given to you for testing.
      ii. The first line that has the number of manager levels, so total number of org chart levels would be number of manager levels + 1 (plus one leaf node level)
         - E.g., managerlevels=5
      iii. The second line has the numbers of max children for the manager levels, using the following line as an example:
         - 3 2 4 5 2
         - The root node (level 0) can have 3 children the most.
         - The level 1 node can have 2 children the most.
         - The level 2 node can have 4 children the most.
         - And so on and so forth
      iv. The rest lines in the orgchart text file should have employee ID paths for building the org chart, one employee ID path per line:
         - You can assume an employee ID always starts with e_, followed by a series of **single digit** integers separated by an underscore, e.g., **e_0_2_3_4**
         - You may have **duplicated** employe ID paths from the org chart text file, but that is ok, if any employee is already added to the org chart, you would just skip adding it.
   b. The **second argument** specifies the file path to the text file that provides employee ID paths for testing (one employee ID per line)
      i. A **testfile.txt** is given for your testing.
      ii. **testCorrectOutput.txt** has the **correct** output for execution using **orgchart.txt and testfile.txt** given to you **(see below)**
   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.

d. **IMPORTANT:** The orgchart.txt and testfile.txt files are given in the **Unix TEXT file** format which has a line feed (**LF**) at the end of each line. **Do NOT** open and save them into **Windows** TEXT file format which would add one more character carriage return (CR, or \r) at the end of each line, making it CRLF. This could **throw off** your logic in reading and processing those files.

4) **Program execution** and **required output:**
   a. Using the following command line **as an example**:
      i. ./countEmployees orgchart.txt testfile.txt
   b. Your program would first read the information from the **orgchart.txt** to build the org chart as elaborated above.
   c. It would then read the employee ID paths in **testfile.txt**, in the **order** of how they appear in the file, and **for every line read in from testfile.txt**:
      i. **Search, count** the number of employees under the org chart starting from that employee ID path, and print the employee ID path followed by a space then the count, one employee ID path per line, e.g.:
         - e_0_2_3 4
         - The printed line above means that starting from e_0_2_3, there are 4 employees added to the org chart, including e_0_2_3.
         - If the node at the employee ID path is not inserted yet, the count would be zero.
   d. See **testCorrectOutput.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 **testCorrectOutput.txt.** You are encouraged to come up with your own test files for testing your code more.
   e. **REMEMBER**: **Do NOT** print the output to a file, print to the **standard output; otherwise, autograding will fail.**

Programming and testing:

- Please refer to C/C++ programming in Linux / Unix page.
- You may use **C++ 11** standard for this assignment, see appendix below and the sample Makefile.
- We strongly recommend you set up your local development environment under a Linux environment (e.g., Ubuntu 18.04 or 20.04, or CentOS 8), develop and test your code there first, then port your code to Edoras (e.g., filezilla or winscp) to compile and test to verify. The gradescope autograder will use a similar environment as Edoras to compile and autograde your code.

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, e.g., ./countEmployees orgchart.txt testfile.txt > testoutput.txt)

- Single Programmer Affidavit with your name and RED ID as signature, use **Single Programmer Affidavit.docx** from Canvas Module Assignments.
- **Number of submissions:**
    a. Please note the autograder counts the number of your submissions. For this assignment, you will be allowed **99** submissions, but **future assignments will be limited to around 10 submissions.** As stressed in the class, you are supposed to do the testing in your own dev environment instead of using the autograder for testing your code. It is also the responsibility of you as the programmer to sort out the test cases based on the requirement specifications instead of relying on the autograder to give the test cases.

If you are a waitlisted student, please see email sent to you for the gradescope entry code (or email TAs), then enroll yourself to the gradescope course for submission.

Grading

Passing 100% autograding 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 on your code appropriately. Code with no or minimal comments are automatically lowered one grade category.
- Design and implement clean interfaces between modules.
- Meaningful variable names.
- NO global variables.
- NO hard code – Magic numbers, etc.
- Have proper code structure between .h and .c / .cpp files, do not #include .cpp files.

Test data including one correct program output are given to you, **any hardcoding** to generate the correct output without implementing the 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.

- Plagiarism detection generates similarity reports of your code with your peers as well as from online sources. It would be purely based on similarity check, two submissions being like each other could be due to both copied from the same source, whichever that source is, even the two students did NOT copy each other.
- We will also include solutions from the popular learning platforms (such as Chegg, github, etc.) as well as code from **ChatGPT** (see below) 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 will also be reported for plagiarism.

- Note the provided source code snippets for illustrating proposed implementation would be ignored in the plagiarism check.

The Center for Teaching & Learning and Instructional Technology Services also shared information concerning the rapidly evolving impact of artificial intelligence (AI) within academia — e.g., ChatGPT, etc.

SDSU's Center for Student Rights and Responsibilities have officially added plagiarism policies of using AI generated content for academic work, as put below:

- "Use of ChatGPT or similar entities [to represent human-authored work] is considered academic dishonesty and is a violation of the Student Code of Conduct. Students who utilize this technology will be referred to the Center for Student Rights and Responsibilities and will face student conduct consequences up to, and including, suspension."

Appendix

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 orgchart.h and orgchart.cpp for tree implementation, and countOrgEmployees.cpp having the main function that orchestrates the program flow.
   - CXX is the variable specifying the C++ compiler as g++ (note autograder uses g++ compiler version 7, Edoras has g++ 4.8.x installed)
   - CXXFLAGS (-std=c++11 -Wall -g3 -c) 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++). -g3 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 -c`

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 -Wall -g3 -c orgchart.cpp
g++ -std=c++11 -Wall -g3 -c countOrgEmployees.cpp
g++ -o countEmployees orgchart.o countOrgEmployees.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.

c. Use “make clean” to delete compiled object code and the executable.

Parse command line arguments

Suppose you put the main program code in a file countEmployees.cpp (C++) or countEmployees.c (C). It should contain the **main(int argc, char **argv)** function. Implementation tips are below.

In the signature of the main(**int argc, char **argv**) function:

- **argc** gives the number of command line arguments of the program including the starting executable name, using
  ./countEmployees orgchart.txt testfile.txt
  as an example, **argc would be 3**
- **argv** contains all command line arguments starting from the executable name, using the above command line execution example:
  - **argv[0]** would be ./countEmployees
  - **argv[1]** would be the file path to the orgchart.txt
  - **argv[2]** would be the file path to the testfile.txt.

Reading a text file line by line

- In C++, use std::ifstream and std::getline.
  - std::ifstream istream(filepath); // open file for reading
    // iterate over text file line by line
    std::string line;
    while (std::getline(istream, line))
- In C, use FILE *fp = fopen(“filename”, "r"), then use fgets(line, sizeof(line), fp) to read each line to a char buffer.