ST. CHARLES COMMUNITY COLLEGE

CPT-281 - Introduction to Data Structures with C++

Team Project 3 (30 Points)

Teamwork Guideline

- Each team will complete either **Project 3A** or **Project 3B**.
- Allowed programming language: **C++**
- Source code should be stored and well maintained in a GitHub project repository. Every team member should contribute to the repository. The repository should be able to be downloaded as a Microsoft Visual Studio project.
- Project report should be a `.doc`, `.docx` or `.pdf` file that is stored in the GitHub project repository. The report should include the following sections:
  1) There should be a **cover page** that shows the project name and all the team members' names. The cover page should be designed professionally.
  2) Show how the system is designed by your team. You need to explain your design, list all data structures you choose to implement the system and explain the role of each data structure.
  3) Draw a **UML class diagram** for your system (please study UML by yourself). Clearly show the logic relationship among all classes in the diagram. For example, class `Movie_List` is an aggregation of class `Movie`, which is a derived class of `Media`.
  4) Show at least **two test cases**. Each test case should contain a sequence of input data and operations. You need to give expected output and compare with the actual output.
  5) In a separate page, clearly list each team member's contribution to the project.
  6) Finally, you need to discuss what improvements to the system could be done in future.
- You **cannot** change team or create new team without the instructor's permission. If a team member refuses to cooperate, please first have a conversation with him/her. If it still does **not** work, then please let the instructor know as early as possible. If you tell the instructor just one day before the project's deadline, saying that one team member **never** replied to emails and **never** did the assigned jobs, the instructor can hardly help you and the project grades of all team members will be suffered.
- "Do Not Cooperate" includes (but not limited to):
  1) Refusing to communicate with other team members, e.g., **never** replying to emails (or reply very late), **never** attending team project meetings (or always show up late for 30 minutes).
  2) Refusing to complete his/her assigned jobs on time so that other team members **cannot** work in the next step, since some jobs can start only after some other jobs are done.

3) Sending "completed" code to others that does **not** compile or does **not** make any sense. Anyone **must** test the completed code before sending to other team members to make sure that the code works properly.

4) Being rude (or in a very unprofessional manner) to other team members, e.g., always having lots of excuses for **not** finishing the assigned jobs on time.

- Source code's value is 20 points; project report's value is 10 points. In principle, all team members will receive the same project grade. However, if the grader believes that a team member did much less job than others, the team member will receive a lower grade than others. If the grader believes that a team member did **nothing** (or almost nothing) in the project, the team member will receive **zero** credits for the project.
- Every team member **must** do some coding job. You **cannot** let a team member writing documents only without doing any coding work.
- Some grading policies:

  1) The grader will download and run your source code. If your code does **not** compile, you will lose at least 20 points and your code will **not** be further graded.

  2) At least 20% of your code should be comments. All variable names, function names, and class names should make good sense. You need to let the grader put the least effort to understand your code. The grader will take off points, no matter whether your code passes all the test cases, if he/she has to put extra unnecessary effort to understand your code.

  3) All the files and folders need to be well organized in the repository. There should be **no** useless files or useless pieces of code.

  4) All the diagrams in your project report **must** be nicely cropped, clean and clear, with **no** unnecessary margins, watermarks, logos, or backgrounds.

- All your work (e.g., source code, project report) should be stored and well maintained in the GitHub project repository. On Canvas, please only submit the URL of your project repository. One submission per team.

Project 3A - Binary Tree Infix Expression Parser

Using binary expression trees, write an infix expression parser. Here are a few examples of expressions your program should parse and evaluate:

Expression	Result
`1+2*3`	`7`
`2+2^2*3`	`14`
`1==2`	`0  // Booleans will be converted to 0 (false) or 1 (true).`
`1+3 > 2`	`1`
`(4>=4) && 0`	`0`
`(1+2)*3`	`9`
`2%2+2^2-5*(3^2)`	`-41`

Technical Requirements

- (Weight: 40%) Your parser should parse an infix expression that supports the following arithmetic and logical operators with the specified precedencies:

Operator	Precedence	Example
`^  // Power`	7	`2 ^ 3  // 8`
`*, /, %  // Arithmetic`	6	`6 * 2  // 12`
`+, -  // Arithmetic`	5	`6 - 2  // 4`
`>, >=, <, <=  // Comparison`	4	`6 > 5  // 1 (true)`
`==, !=  // Equality Comparison`	3	`6 != 5  // 1 (true)`
`&&  // Logical And`	2	`6 > 5 && 4 > 5  // 0 (false)`
`||  // Logical Or`	1	`1 || 0  // 1 (true)`

- (Weight: 30%) Parse an expression given in a string format. Your program should be flexible with the given expressions. For instance, "**1+2**" is the same as "**1 + 2**". The user should **not** worry about writing the spaces between operands and operators.
- (Weight: 20%) Evaluate the given expression efficiently.
- (Weight: 10%) Your **`main()`** program should read expression(s) from an input file, then output the evaluation result(s) to the console.

Facts and Assumptions

- You may assume that **all operands are integers**. However, an operand may contain more than one digit.
- For divisions, you need to get the integer result. For example, "**`3 / 2`**" should be evaluated to 1 instead of 1.5.
- The result of a comparison is a **boolean**. However, a boolean can be converted to an integer according to the logic of **`true == 1`** and **`false == 0`**. Also, an integer can be converted to a boolean according to the logic that a number equal to zero is **`false`**; otherwise, it is **`true`**. Examples are given below:

Expression	Evaluation Result
`(3 + 4) || 1`	`1`
`(2 > 3) - 2`	`-2`
`5 ^ 2 % 7 && (4 - 4)`	`0`

- Take a look at the algorithm in the following link for inspiration. The algorithm constructs a binary expression tree from a postfix expression. In this project, however, you are required to build the tree from an infix expression.

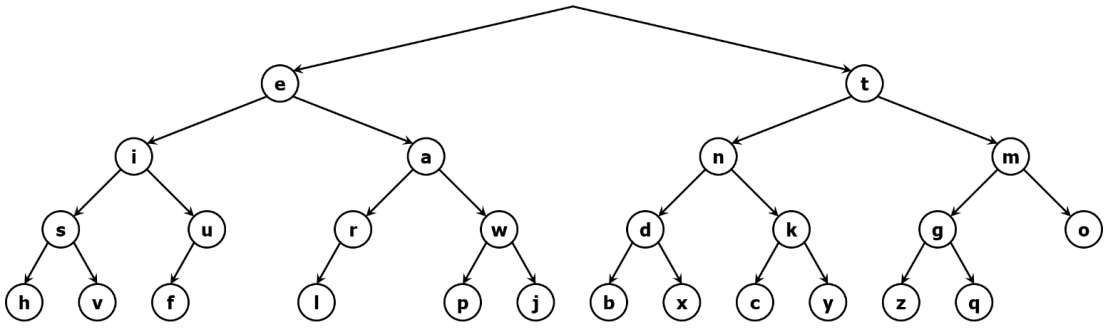
  https://en.wikipedia.org/wiki/Binary_expression_tree#Construction_of_an_expression_tree

- You can assume that the users always input valid expressions. However, if the input expression to be evaluated is "**`3 / (6 * 5 - 30)`**", your program should prompt the user that there is a divide-by-zero error, instead of a crash.
- You **must** decompose the project into different classes (Object-Oriented Programming). You **cannot** write all your code in a single file.

Morse code (see the table below) is a common code that is used to encode messages consisting of letters. Each letter consists of a series of dots and dashes. For example, the code for the letter `'a'` is "•–" and the code for the letter `'b'` is "–•••".

Letter	Code	Letter	Code	Letter	Code	Letter	Code
a	•–	h	••••	o	–––	v	•••–
b	–•••	i	••	p	•––•	w	•––
c	–•–•	j	•–––	q	––•–	x	–••–
d	–••	k	–•–	r	•–•	y	–•––
e	•	l	•–••	s	•••	z	––••
f	••–•	m	––	t	–		
g	––•	n	–•	u	••–		

Store each letter of the alphabet in a node of a binary tree of depth 4. The root node is at depth 0 and stores no letter. The left node at depth 1 stores the letter `'e'` (code "•") and the right node stores the letter `'t'` (code "–"). The four nodes at depth 2 stores the letters with codes "••", "•–", "–•" and "––". To build the tree (see the tree below), read a file in which each line consists of a letter followed by its code. The letters in the file are in alphabetical order, **not** the order that is convenient to build the tree. To find the position for a letter in the tree, scan the code and branch left for a dot and branch right for a dash. Encode a message by replacing each letter by its code symbol. Then decode the message using Morse code tree. Make sure you use a delimiter symbol between coded letters.



Technical Requirements

- (Weight: 30%) Write a method that builds the Morse code tree shown above. The information of the tree (the letters and the codes) is stored in a file. You can find the file by clicking the link below.

  https://drive.google.com/uc?export=download&id=1yoGcPU_ipgOkmB1AmRM1AhBUgSo1JwNS

  Notice that the file is **not** been laid out in convenient order for building the tree.

- (Weight: 30%) Your system should be able to decode a message using the Morse code tree that you built. For example, decoding **"–•• ––•"** results in **"dg"**. The text above briefly explains how you can do that. Notice that between the symbols (dots and dashes) is a space. The space is a delimiter that separates the codes for letters.
- (Weight: 30%) Your system should also encode a message. For example, encoding **"ac"** results in **"•– –•–•"**. Please note that you **must** use the Morse code tree you built to encode a message. You **cannot** work around this, creating maps, lists, or hash table to encode text.
- (Weight: 10%) Your **`main()`** program should let the user input a string from the keyboard. Then, show the encoded message to the user. Then, decode the message and show the result to the user.

Facts and Assumptions

- You may assume that the character delimiters are simply spaces.
- You may assume the user input string has only lowercase letters with **no** spaces.