Lab Mock Exam

Instructions for Lab Submission:

1. File Naming and Submission:
   a. Submit your solution as a zip file named **lab_mock _<ERP>.zip**, where *<ERP>* is your ERP number. For example, if your ERP number is *12345*, the file should be named *lab_mock _12345.zip*.
   b. Ensure that the zip file:
      i. Contains the following directories:
         1. ***headers/***: All header files, named as ***<class/struct name>.hpp***
         2. ***src/***: All implementation files named as ***<class/struct name>.cpp***
         3. ***main/***: All files with main function for each task named as ***Task<number>.cpp***
      ii. Does not include unnecessary files like executables.
   c. Do not submit ***TEXT*** files.
2. Code Organization:
   a. Each class or struct used in a task must have its declaration in a *.hpp* file (in the ***headers/*** directory) and its implementation in a *.cpp* file (in the ***src/*** directory). The logic and function calls should be written in a separate main file (in the ***main/*** directory).
   b. Ensure that the code structure is modular and adheres to the principles of good software development (e.g., separation of concerns).
3. Code Neatness:
   a. Ensure your code is well-formatted and easy to read. Use proper indentation and meaningful variable names.
   b. Include appropriate comments to explain the logic where necessary (especially for functions and complex sections of the code).
   c. **Add clear explanations where required(or specified)** to make the logic of your program easy to follow.
4. Code Compilation and Functionality:
   a. Ensure your code compiles without errors or warnings. You should be able to compile and run your code on any system with a standard C++ compiler (e.g., GCC, Clang, MSVC).
   b. Double-check that the functionality of the program meets the requirements outlined in the task.
5. Additional Instructions:
   a. Do not use any libraries or features that have not been covered in the course material, unless specifically instructed.
   b. Make sure your code adheres to the principles of good software development (e.g., modular functions, minimal repetition, and no hardcoding).
   c. If you encounter any issues or require clarifications, reach out before the submission deadline.
6. Deadline:
   a. Ensure that you submit your lab on time, before the deadline. Late submissions will not be accepted
   b. Double-check that the zip file is correctly named and contains all necessary files before submitting.
7. Plagiarism Policy:
   a. The work submitted must be entirely your own. Any code or content copied from others (including online sources, classmates or AI) will result in an automatic zero for the task.

Tasks

Task 1:

You are required to implement a class named **WordData** that is responsible for parsing a given text file and internally storing the word-frequency data in an efficient data structure. The structure you choose must support **O(log n) or better** time complexity for operations like insertion, deletion, and lookup and must be implemented yourself.

When a *WordData* object is constructed, it must take the path to a **.txt** file and automatically read and parse the file, storing each word along with its frequency. Words should be treated in a **case-insensitive** manner (i.e., Apple and apple count as the same word), and any punctuation or non-alphabetic characters must be excluded from the parsing.

The internal data structure must not be exposed publicly. All interaction must occur through the class's public interface.

Required Public Methods

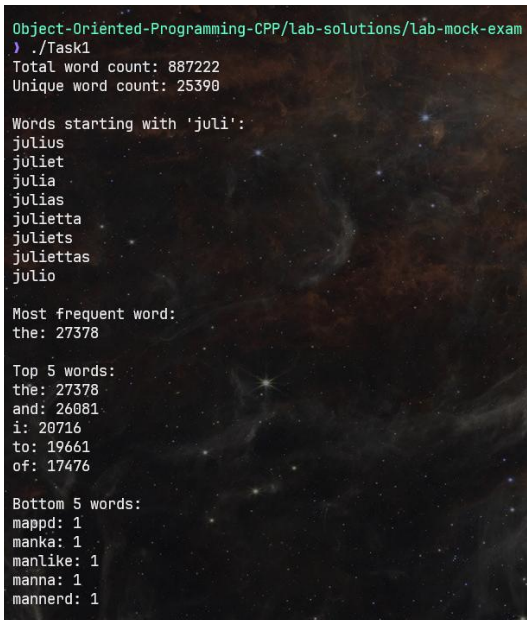
- **int getFrequency(const std::string& word) const**
  Returns the frequency count of the specified word. If the word is not present in the corpus, it should return 0. The input word is case-insensitive.
- **bool containsWord(const std::string& word) const;**
  Returns true if the specified word exists in the corpus; otherwise, returns false. Also case insensitive
- **int getTotalWordCount() const;**
  Returns the total number of **words** in the corpus, counting **repetitions**. For example, if "hello" appears 5 times and "world" appears 3 times, the total word count is 8.
- **void exportToFile(const std::string& path) const;**
  Exports all words in the corpus along with their frequency counts to a .txt file. Each line of the file should contain a word followed by its count, separated by **tabs**. The format should be:

word<TAB><TAB>count

Words should be exported in **lexicographical order** (case-insensitive). Addtionally your class should provide the following querying methods

- **starts_with**: Returns all words beginning with a given prefix.
- **most_frequent**: Returns the word(s) with the highest frequency.
- **longest_word**: Returns the longest word(s) in the corpus.
- **top_k**: Returns the top K most frequent words.
- **bottom_k**: Returns the K least frequent words.

shakespeare.txt, and ***small-text.txt*** can be used as test corpora.



```
Object-Oriented-Programming-CPP/lab-solutions/lab-mock-exam
❯ ./Task1
Total word count: 887222
Unique word count: 25390

Words starting with 'juli':
julius
juliet
julia
julias
julietta
juliets
juliettas
julio

Most frequent word:
the: 27378

Top 5 words:
the: 27378
and: 26081
i: 20716
to: 19661
of: 17476

Bottom 5 words:
mappd: 1
manka: 1
manlike: 1
manna: 1
mannerd: 1
```

Task 2:

You will implement a generic expression evaluator that models mathematical expressions as a chain of operations, each encapsulated in its own class. The evaluator takes an initial input and applies a sequence of transformations, mimicking the structure of nested expressions like f(g(h(x))). Each operation is represented as an object, allowing dynamic chaining through a custom singly linked list.

Operations are defined using a templated abstract base class, with specific operations implemented in derived classes. Some operations, particularly costly ones like exponentiation or trigonometric functions, must implement caching to avoid redundant computation. Others may adjust behavior based on previous inputs. The evaluator applies each operation in sequence by traversing the list and transforming the input step by step.

1. Base Class: Operation

You must define a templated abstract base class **Operation<T>** that declares the interface for all operations.

- **T apply(T input) const:** Applies the operation to the input and returns the result.
- **std::string name() const:** Returns a string that identifies the operation.

This class represents any transformation on a value of type T. It serves as the foundation for all specific operations.

2. Derived Operation Classes

You must implement the following derived classes, all templated on type T:

- **Add<T>**
  - Adds a fixed constant to the input.
  - Constructor accepts a constant value.
  - Example: Add(5) turns input x into x + 5.
- **Multiply<T>**
  - Multiplies the input by a fixed constant.
  - Constructor accepts a constant value.
  - Example: Multiply(3) turns x into x * 3.
- **ExpensivePower<T>**
  - Raises input to a fixed exponent using repeated multiplication (not std::pow).
  - Example: ExpensivePower(3) turns x into x * x * x.
  - This operation is intentionally computationally expensive to justify caching in future extensions.
- **CachedSine<T>**
  - Computes the sine of the input using **std::sin**, but caches the last input and output.
  - If the same input is applied again, the cached result is reused instead of recomputing.
  - Use mutable state to implement caching within a const apply() function.
- **CachedAdd<T>**
  - Like Add, but instead of adding a fixed constant, it adds the result of the last apply() call.
  - Maintains an internal cache of the last computed result.
  - Example: if last result was 7, next input x becomes x + 7.
- **CachedMultiply<T>**
  - Like Multiply, but multiplies the input by the cached result from the last evaluation.
  - If no cached value exists (i.e., on first call), treat the cache as identity (i.e., multiply by 1).

Each of these classes must override both **apply()** and **name()** meaningfully and may internally store constants, cache values, or state as required.

3. OperationNode

Define a templated struct OperationNode<T> to represent a node in a singly linked list.

- Contains a pointer to an Operation<T> object.
- Contains a pointer to the next OperationNode<T>.

Each node wraps a single operation and is part of a chain forming the complete expression.

4. ExpressionEvaluator<T>

This class manages the overall expression pipeline and handles the evaluation logic.

- **ExpressionEvaluator():** Initializes an empty operation list.
- **void addOperation(Operation<T>* op):** Appends a new operation to the end of the list.
- **T evaluate(T input) const:** Traverses the operation chain and successively applies each operation to the value.
- **void printPipeline() const:** Prints the name of each operation in the chain in order.
- Destructor must clean up all heap-allocated operations and list nodes.

The evaluator must be templated on T, allowing it to work with various types such as int, float, or double.

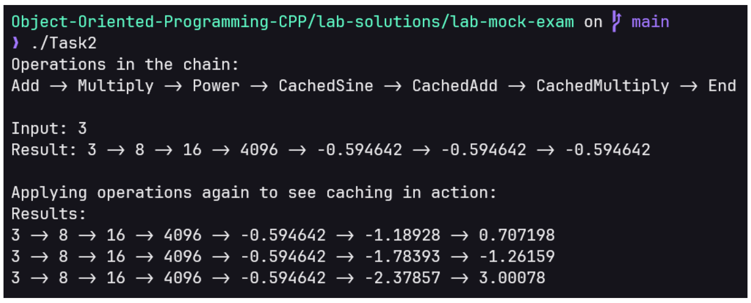
5. Example Execution

An example usage would involve constructing an evaluator as follows:

- Create an ExpressionEvaluator<double>.
- Add the following operations:
  - Add(5)
  - CachedSine()
  - ExpensivePower(2)
- Evaluate the expression for input x = 3.14.

This would result in the following transformations:

- Add: 3.14 + 5 → 8.14
- Sine: sin(8.14) → s
- Power: s * s → final output



```
Object-Oriented-Programming-CPP/lab-solutions/lab-mock-exam on main
❯ ./Task2
Operations in the chain:
Add → Multiply → Power → CachedSine → CachedAdd → CachedMultiply → End

Input: 3
Result: 3 → 8 → 16 → 4096 → -0.594642 → -0.594642 → -0.594642

Applying operations again to see caching in action:
Results:
3 → 8 → 16 → 4096 → -0.594642 → -1.18928 → 0.707198
3 → 8 → 16 → 4096 → -0.594642 → -1.78393 → -1.26159
3 → 8 → 16 → 4096 → -0.594642 → -2.37857 → 3.00078
```

Subsequent evaluations with the same input to CachedSine would use the cached sine value, avoiding recomputation.

Notes:

- The Operation<T> interface must remain general and extensible for further derived operations to be added without modifying existing code.
- Implement const-correctness and avoid unnecessary copies wherever possible.