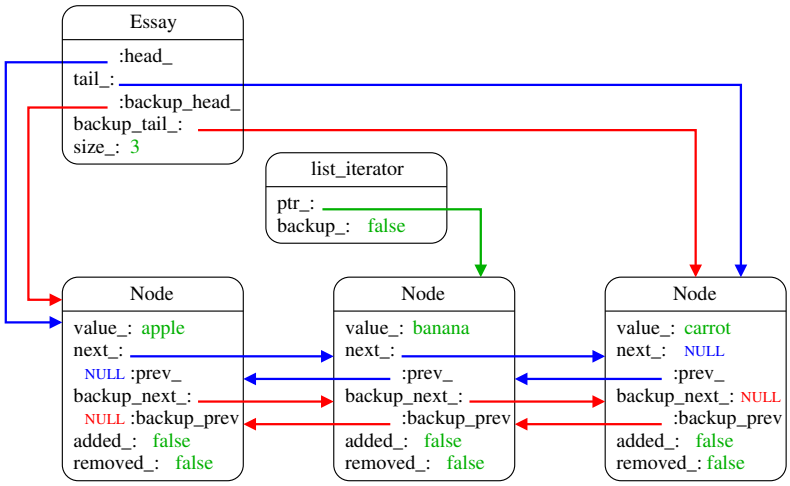
CSCI-1200 Data Structures — Fall 2019
Homework 5 — Backup Essay

In this assignment you will develop a data structure to store a doubly linked list of nodes storing STL strings that represent the words in an essay. The user of this `Essay` class can perform typical editing actions to insert or erase one word at a time anywhere in the document. They can also "cut & paste" or *splice* a sequence of words from one location of the document to another. The exciting new feature for the `Essay` class that makes it different from STL `list` and the `dslist` class we saw in Lecture 10 is that we can *backup* the current version of the essay. As we continue to make further edits, the backup information is preserved, and we can access it through iterators. If we choose to *revert* the essay to the backup version, we effectively undo all actions performed since the backup. Note that we only store a single backup, we do not store the full history of edits. *Please read through the entire handout, and study the provided test code in* `main.cpp` *before beginning your implementation.*

The Essay Data Structure

To implement the `Essay` data structure, we will add the following member variables to the `Node`, `list_iterator` and `Essay` classes from Lecture 10. Note that we have renamed the `dslist` class to be the `Essay` class. Also note that the Essay class is *not* templated. The value in each `Node` is an STL `string`.

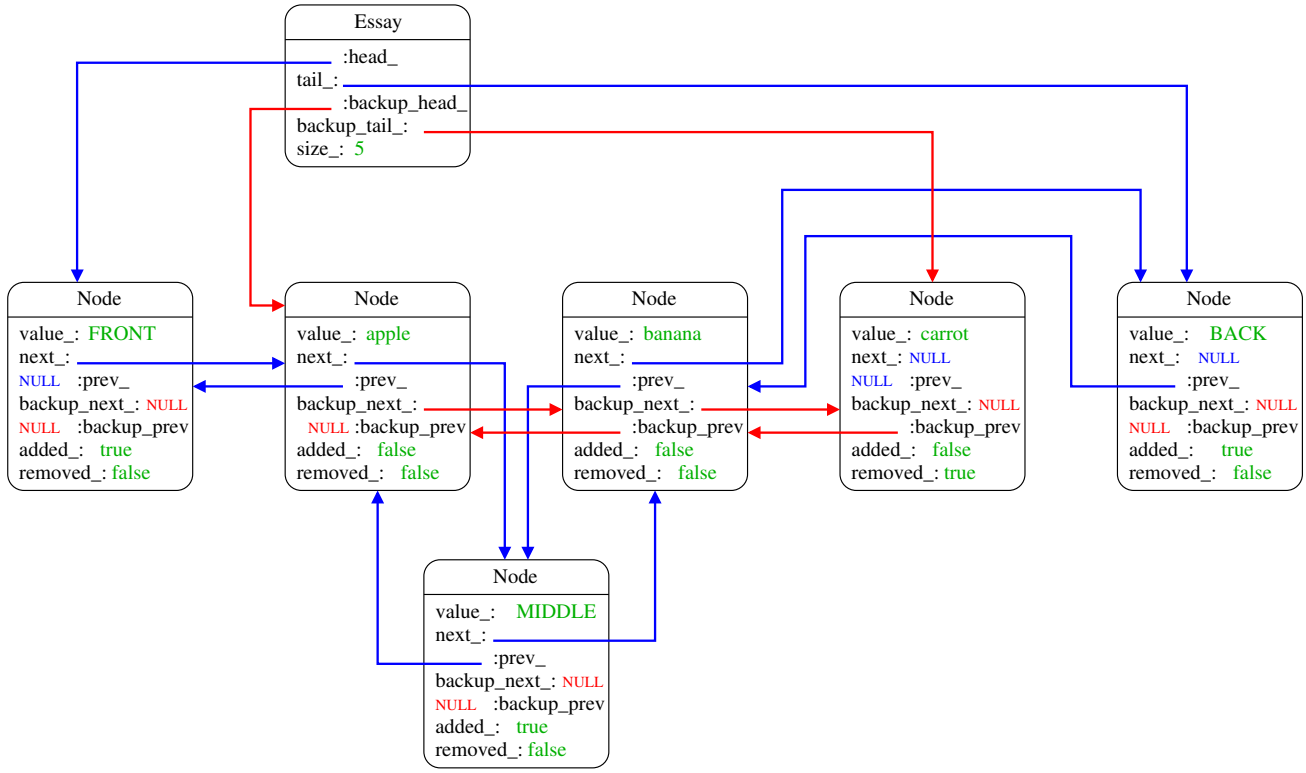


Each `Node` object has 2 additional pointers, `backup_next` and `backup_prev`, that record the position of this word within the backup version of the Essay. Additionally, to help determine when a `Node` should be safely deleted, we add the boolean variables `added_` and `removed_`. `added_` is true if the `Node` has been added (via `push_front`, `push_back`, or `insert`) since the backup. Similarly, `removed_` is true if the `Node` has been removed (via `pop_front`, `pop_back`, or `erase`) since the backup. The `list_iterator` for the `Essay` class must now support traversing the `Node`s in either the current version of the essay (blue links) or the backup version of the essay (red links). Therefore, we add a boolean `backup_` to the iterator that is false if we are stepping through the nodes in the current essay, and true if we are stepping through the backup essay. Finally, the `Essay` class has 2 additional pointers `backup_head_` and `backup_tail_` that store the first and the last nodes of the backup version of the essay. The Essay structure can pack *two different orderings* of doubly-linked lists within it. When the document is lengthy and the edits since the backup are small, the structure will use less memory than storing the two separate linked lists!

Provided Testing Framework

To further specify the required interface for the `Essay` class and to aid in your development and testing, we provide a `main.cpp` file with sample test cases. The diagram below illustrates the example from the

`test_backup_and_revert` function. We backed up the essay when it was just 3 words: `apple banana carrot`. Then we did a `push_front`, `push_back`, `insert` and `erase`. The essay now reads: `FRONT apple MIDDLE banana BACK`. When we perform a revert, we will permanently delete the 3 nodes with `added_` = true and set all pointers back to the state on the previous page. Note that immediately after a `backup` or `revert` operation: `next_` == `backup_next_`, `prev_` == `backup_prev_`, and `added_` and `removed_` are both false in every node in the structure. Note that `added_` and `removed_` are never both *true* in the same node.



Your task is to implement the `Node`, `list_iterator`, and `Essay` classes. Note that because `Essay` is not templated, you will write two files: `essay.h` and `essay.cpp`. You are encouraged to comment and uncomment the provided tests in `main.cpp` as you work through the assignment. But your final submission for this assignment should not change the `main.cpp` file except to add your own test cases within the `student_tests` function. You will be graded on the completeness of the additional tests you write to explore the "corner cases" of the required member functions of `Essay` demonstrating that your implementation is complete and robust. You should also create tests of the `Essay` class copy constructor, assignment operator, and destructor.

Additional Requirements, Hints, and Suggestions

You may not use lists or vectors or other STL containers in this assignment. Instead, you will be manipulating the low-level custom `Node` objects, and the pointers that connect each `Node` to other `Node`s in the structure. You must implement the data structure exactly as diagrammed, with the specified member variables. The only exception is an allowed modification to facilitiate decrement of the `end()` and `backup_end()` iterators.

Submitty will again be configured to use Dr. Memory to check your program for memory errors and memory leaks. Be sure to use Dr. Memory or Valgrind on your local machine as you develop to catch these problems early. Use good coding style when you design and implement your program. Be sure to make up new test cases and don't forget to comment your code! Please use the provided template `README.txt` file for any notes you want the grader to read. You must do this assignment on your own, as described in the "Collaboration Policy & Academic Integrity" handout. If you did discuss this assignment, problem solving techniques, or error messages, etc. with anyone, please list their names in your `README.txt` file.