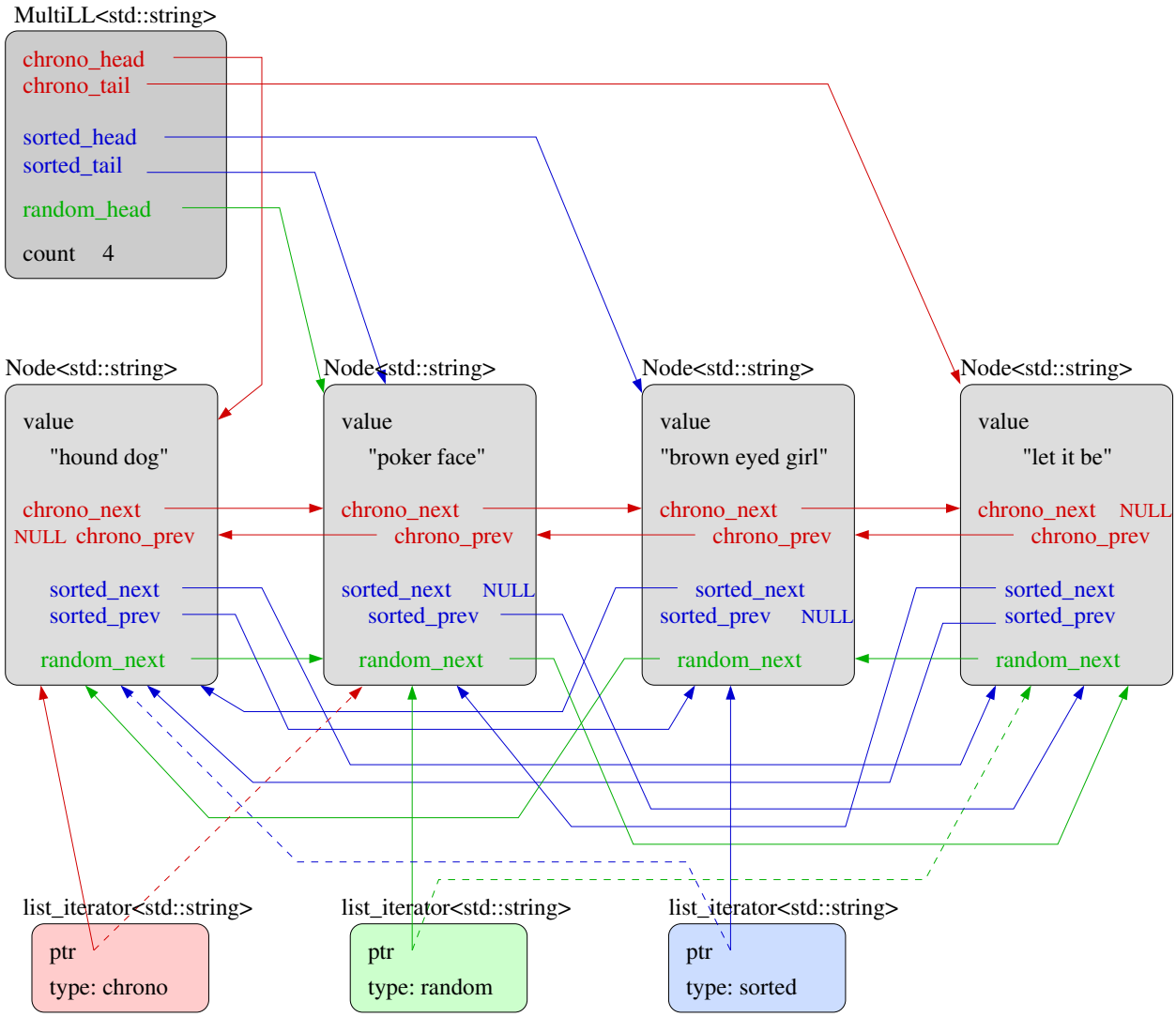
CSCI-1200 Data Structures — Fall 2020
Homework 5 — Multi-Linked Lists

In this assignment you will build an extension of the basic linked list data structure called a *multi-linked list*. Our particular variation for this homework has elements of singly-linked, doubly-linked, and circularly linked lists. This style of data structure can be advantageous is a variety of applications; for example, in storing and organizing a musical song database that can be iterated/viewed/played in chronological (insertion) order, in alphabetical (sorted) order, and in random order (a.k.a. "shuffle" mode). You're welcome to read more about multi-linked lists on Wikipedia (but you're not allowed to search for, use, or study in detail any code or pseudocode that might be available online!). **Please read the entire handout before beginning your implementation.**

To complete the assignment you will modify the helper classes (`Node` and `list_iterator`) as well as the main class, `dslist`, which will be renamed "`MultiLL`". Below is a picture of the relationships between the classes (compare this to the picture of the `dslist` class from Lecture 10):



That's a whole lot of pointers!!! Each `Node` object has been expanded to include several new pointers to its fellow nodes. In addition to the basic doubly-linked list pointers (renamed `chrono_next` and `chrono_prev` and drawn in red), each node contains pointers for a second doubly-linked structure in which the nodes are

linked in alphabetical order (through the blue `sorted_next` and `sorted_prev` pointers). Finally, a third order of the nodes is stored with a *singly-linked, circular* chain through the `random_next` pointers (in green). Note that the random ordering is re-created (*re-randomized*) each time a random iterator is requested/attached to the list.

These three orderings are maintained by the top level `MultiLL` object, and are accessed through appropriately named private `head` and `tail` pointers. Furthermore, because there are three different orderings, the iterator objects that attach to the `MultiLL` also include a `type` variable that is used when incrementing or decrementing the iterator to specify which of the three orderings should be followed. This type can be implemented as a custom `enum` type or with boolean values or an integer (your choice). In the diagram above, we show three iterators of different types attached to the list, as they would be initialized with the corresponding `begin_chronological()`, `begin_sorted()`, and `begin_random()` function calls. The dashed lines illustrate where these iterators will move if they are pushed forward with the `++` increment operator (called separately on each iterator).

`Node`s can be inserted into the `MultiLL` using the `add` function. We specify that the new node is *always* inserted at the tail end of the chronological ordering, and the node is *automatically* inserted into the sorted list to maintain the consistent alphabetical ordering. The other common methods of adding elements to a linked list (e.g., `push_front`, `push_back`, and `insert`) are *not available* in our `MultiLL`.

Just like regular linked lists, the `MultiLL` supports speedy `erase` operations of any `Node` from the beginning, middle, or end of the list. Of course, this involves a lot of pointer rearrangement! The `erase` operation is passed an iterator (of one of the three types), and after carefully removing the `Node` that the iterator points to, it returns an iterator of the same type, that points to the `Node` that is "next" in the order specified by that type.

Randomness

Each time the `begin_random()` member function is called to attach a new iterator to the `MLList`, the singly-linked circular chain of `random_next` pointers is re-randomized. The new ordering should be a proper random distribution; that is, each element in the list is equally likely to be chosen as the new head for the random ordering. And each remaining element is equally likely to be chosen as the "next" node after the head, etc. Finally, the last node in random order points back to the head node. All elements should appear in the circular list. Note that because this is a circular list, incrementing the random iterator can be done infinitely. The random order is generated once for the initial loop and the nodes simply repeat this same order on all subsequent loops.

Randomness is simulated by a computer using a *pseudo-random number generator*. STL includes a library of tools to generate random numbers, we provide a simple example use of this library to generate random integers (`random_example.cpp`). It will be helpful to test your program both with a deterministic, fixed random sequence (which is useful for debugging), and a randomly *seeded* sequence, so that subsequent runs will be different.

Your Tasks

Your new data structure should be a templated class named `MultiLL` and its public interface should support the following member functions: `size`, `empty`, `clear`, `add`, `erase`, `begin_chronological`, `begin_sorted`, `begin_random`, `end_chronological`, and `end_sorted`. The `begin_chronological`, `begin_sorted`, `end_chronological`, and `end_sorted` functions must run in *constant time*. Note that the running time of the `begin_random` function is not specified. The chronological and sorted iterators should support bi-directional movement (i.e., increment `++` and decrement `--` operations) and the random type iterator should support the increment operator. All of these iterator operations should run in *constant time*. Note: You should implement all standard class constructors, the assignment operator, and the destructor and these functions should properly allocate and deallocate the necessary dynamic memory. As with Homework 3, we expect

you to use a memory debugger (Dr. Memory or Valgrind) on your local machine to find and fix all memory errors and leaks. Please ask the instructor or TAs for help if you have trouble installing or using these tools. We will enable the homework server to give you the output from Dr. Memory and to receive full credit from the automatic grading, your code must be memory error and memory leak free.

To get started, we highly recommend that you study and heavily borrow from the `dslist` class that we discussed in Lecture 10. The code is available on the course website. We provide a `main.cpp` file that will exercise the required features of your class, but it *does not include tests for each "corner" case.* You should add your own test cases to the bottom of the `main.cpp` file. Be sure to test every member function each of the constructors, the assignment operator, and the destructor, and exercise the templated class with types other than `std::string`. Discuss in your `README.txt` file your strategy for thoroughly debugging your assignment.

In the implementation of the `MLList` class you *may not* use the STL `list` or the STL `vector` classes (or any other advanced STL container or homemade versions of the above). You may only use the STL `list` and/or `vector` classes to aid in testing and debugging, in the `main.cpp` file.

Big O Notation Analysis

You must follow the memory structure and performance requirements outlined in this handout for the `MLList` class. Overall, you should pay attention to and strive for efficiency when implementing the data structure. However, for the basic assignment you are specifically encouraged to use a simple $O(n^2)$ sorting algorithm such as insertion sort. What is the Big O Notation of each of the member functions in your class? Include your answers and discussion of this analysis in your `README.txt`.

Iterator Invalidation

The interaction of active `MLList` iterators with the `add` and `erase` functions, and the simultaneous use of multiple `MLList` iterators on the same `MLList` object, may cause inconsistencies that are non-trivial, inefficient, and/or impossible to resolve. Think carefully about potential problems that might arise. In your `README.txt` file discuss any restrictions you chose to place on the use of your class; specifically, if the user should be aware of the invalidation of active iterators during the use of specific member functions. Be sure to appropriately justify the need for these restrictions.

Extra Credit

Your Big O Notation evaluation should have uncovered one function that will benefit from an improved sorting algorithm (something better than insertion sort). Which function? Hint: It's not `add`! Caution, your overall class must still meet the performance requirements outlined above. For extra credit, re-implement this function to achieve a significant improvement in performance, correctly identifying the Big O Notation of both your original and revised implementations. Discuss in your `README.txt`.

Additional Information

Use good coding style when you design and implement your program. Be sure to make up new test cases to fully test your program and don't forget to comment your code!

You must do this assignment on your own, as described in the "Collaboration Policy & Academic Integrity" handout. Use the template `README.txt` to list your collaborators and any notes you want the grader to read.