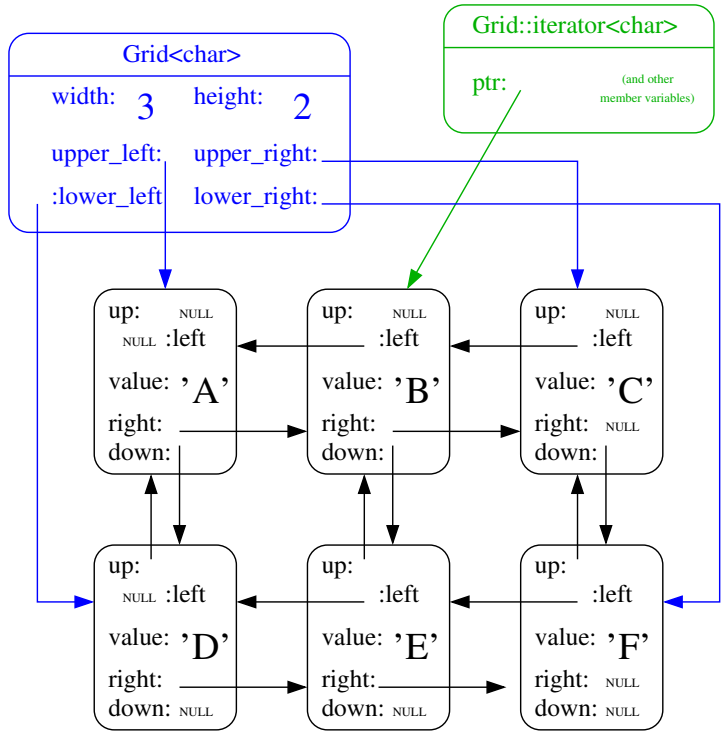
CSCI-1200 Data Structures — Fall 2018
Homework 5 — Linked Grid

In this assignment you will develop a data structure to store a two-dimensional quadruple-linked grid of nodes storing an arbitrary templated type. For example, this structure can be used to store a mathematical matrix of floating point numbers or it can be used to store an image of red/green/blue pixels. However, we note that for many typical matrix or image computations an array or vector with efficient indexing/random access is generally preferable.

While the linked grid data structure you will implement for this homework may be slower than arrays for indexing, it can be significantly more efficient for merging, splitting, and cropping. These operations are useful and important tasks in applications including fluid dynamics (e.g., water or gas), deformable bodies (e.g., soft tissue or soil), cloth simulation, and other scientific and massively-parallel computations. The basic structure we explore for this homework can be extended and generalized to three dimensions for volumetric elements, a.k.a. voxels, or hexahedral or tetrahedral elements, using structured grids or unstructured layouts. *Please read through the entire handout, and study the provided test code in* `main.cpp` *and the sample output before beginning your implementation.*

The Linked Grid Structure

You will implement three classes for this homework: the `Grid` manager class which contains pointers to `Node` objects. To navigate the data in the `Node`s, we use a `GridIterator` object, which is `typedef`'d as `Grid::iterator`. All three classes are templated over a placeholder type `T`. In the diagram below, `T=char`. In your implementation you must follow this diagram exactly, including only these specific member variables. The details of the member variables for the `GridIterator` class beyond the required `ptr` variable is the only section that is left unspecified and open for your design.



In addition to the `value` member variable, each `Node` object has a total of four pointers to its immediately adjacent `Node`s in the structure. On the edges of the grid structure, these pointers are set to NULL.

The `Grid` manager class stores the current width and height of the grid, and four pointers to the corners of the structure, which help alleviate performance concerns for some operations when the size of the grid is quite large. The `Grid` data structure allows the user to `set` or `get` individual values from the `Grid` by row and column indices. Two `Grid` objects can be vertically or horizontally merged using the `stack` and `join` operations. And we can separate a `Grid` into two `Grid`s with the reverse operations `lift` and `chop`. See the provided tests and the sample output for details. Note that the `print` member function is primarily for debugging and only needs to work with "small" types/values can be printed to `std::cout` with `setw(3)`.

Due to the inherent two-dimensional structure, the primary navigation of the basic `GridIterator` will be through four intuitively named functions `right`, `down`, `left`, and `up` instead of the typical operators `++` and `--`. Basic iteration can be started at any of the four corners of the grid with the `begin_upper_left`, `begin_upper_right`, `begin_lower_right`, and `begin_lower_right` function calls. If the iterator walks off the edge of the grid, is equivalent to be `end()` iterator. For extra credit, you can allow the user to return to the previous non-NULL `Node` by reversing the last iteration action.

The grid structure also allows two specialty iteration patterns: *snake* and *spiral*. See the provided sample output. These patterns are launched with the `begin_snake` and `begin_spiral` functions and advanced with the pre- and post- increment `operator++`. *Hint: Save the spiral pattern for last since it's complicated and won't be worth very many points overall.* For extra credit you can also implement the decrement `operator--` and handle decrement of the `end_snake` and `end_spiral` functions.

Provided Testing Framework

To further specify the required interface and to aid in your development and testing, we provide a `main.cpp` file with sample test cases and sample output. You should work through the implementation of this homework in stages, uncommenting the provided tests in `main.cpp` as you work. Note: Because the `Grid`, `GridIterator`, and `Node` classes are templated and heavily intertwined, you will write the class declaration and *all* function implementations for all three classes in a single new file named `grid.h`.

Your final submission for this assignment should not change the `main.cpp` file except to uncomment the tests you have completed and to add your own test cases within the `additional_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 `Grid` demonstrating that your implementation is complete and robust. You should also create tests of the `Grid` class copy constructor, assignment operator, and destructor.

Complexity Analysis

Your implementation of each operation should be streamlined and efficient. In your `README.txt`, give the Big 'O' Notation for each operation, assuming the grid is w nodes wide and h nodes high.

Additional Requirements, Hints, and Suggestions

You may not use arrays or 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 incomplete specification of the member variables of the `GridIterator` class.

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.