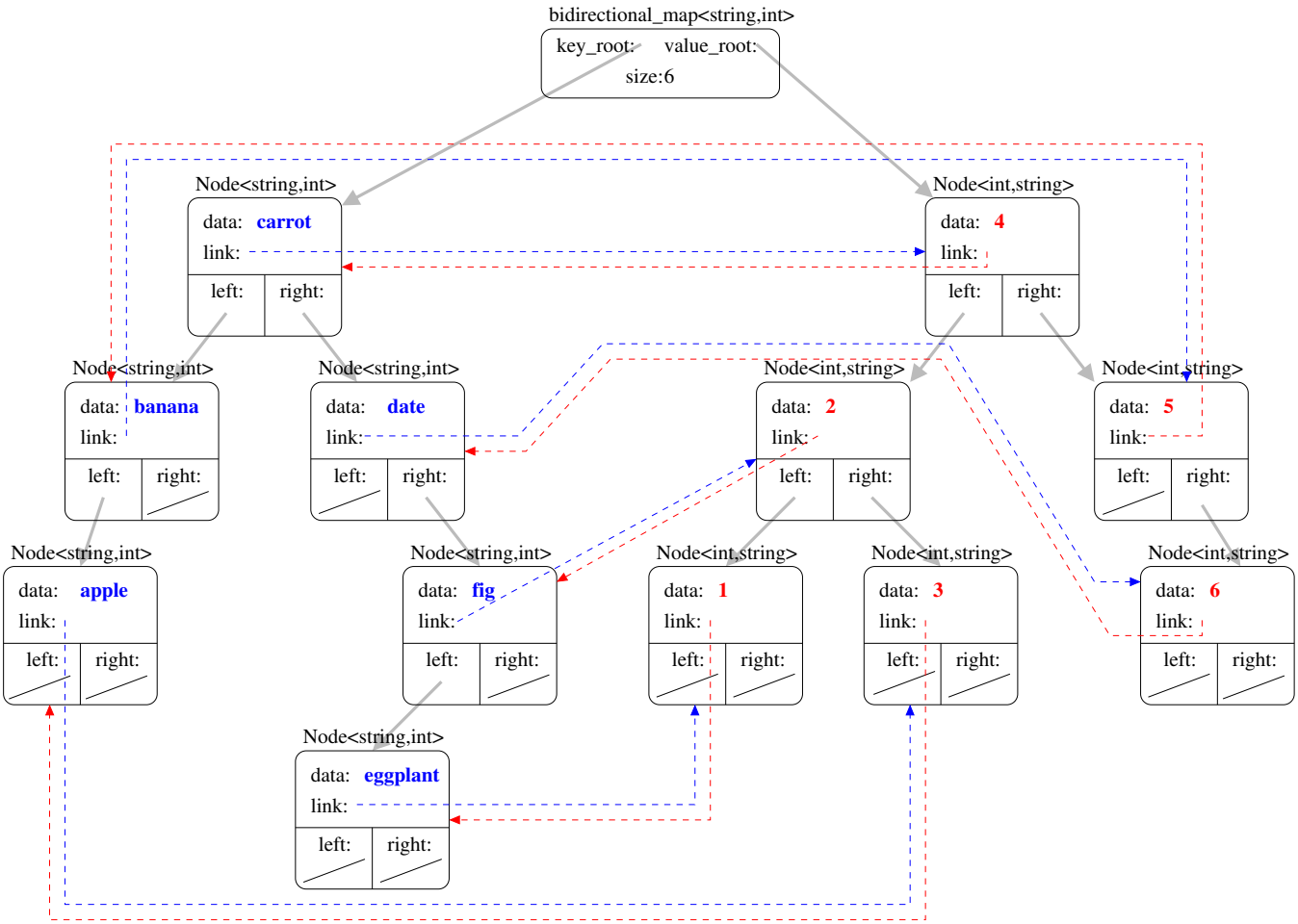
CSCI-1200 Data Structures — Fall 2014
Homework 8 — Bidirectional Maps

In this assignment you will build a custom data structure named `bidirectional_map`. A `bidirectional_map` is similar to an STL `map`, in that it stores the association between *key* and *value* data. Like a `map`, inserting and erasing an association between a key and a value is completed in O(log n), where n is the number of association pairs in the map. Also, like a `map`, lookup by key is O(log n), but additionally, with a bidirectional `map`, lookup by value is also O(log n)!

The typical implementation of an STL `map` stores STL `pair`s of the key and value in a single binary search tree, ordered by the keys. Our `bidirectional_map` will instead be stored in two distinct binary search trees, one storing the keys and one storing the values. The associations are stored as bidirectional *links* between the nodes of the two trees.

The diagram below illustrates the core structure of the `bidirectional_map` data structure:



Note that the `bidirectional_map` is templated over two types, the *key type* and the *value type*. In this example, the key type is STL `string` and the value type is `int`. We are currently storing 6 associations in this structure. Note that in this initial example, the associations are *"one-to-one"*. This means that there are no repeated or duplicate keys *or* values.

Like an STL `map` it is not possible to edit the key of an association, because that edit may disrupt the overall binary search ordering property of the tree. Because the values of our `bidirectional_map` are also stored in their own binary search tree, we must also prohibit editing of the values in a `bidirectional_map`. If the key or the value of an association needs to be changed, that data must be removed from the tree and re-inserted.

Implementation

Your task for this homework is to implement the structure diagrammed on the previous page. We recommend that you begin your implementation by following the structure of the `ds_set` class we studied and worked with in lecture and lab. You will need to make a number of significant changes to the code, but the overall design composed of 3 classes – `Node`, `tree_iterator`, and "manager" class (`bidirectional_map`) – remains the same. Note that each class is now templated over two types rather than just one.

The provided code in `main.cpp` illustrates the basic functionality of the `bidirectional_map` class including the `bidirectional_map` functions: `size`, `insert`, `erase`, `find`, `operator[]`, `key_begin`, `key_end`, `value_begin`, and `value_end`. Study these examples carefully to deduce the expected argument and return types of the functions. The `bidirectional_map` has iterators over both the keys and values, which can be incremented and decremented like regular STL iterators. Also, each iterator's `follow_link` function can be used to obtain an iterator pointing to the associated data in the other half of the structure. You will also write a `print` function to help your implementation and debugging.

As this is a class with dynamically allocated memory, you will also need to implement the default constructor, copy constructor, assignment operator, and destructor. The provided code in `main.cpp` does not thoroughly test these functions, so you must write your own test cases. The homework server will compile and run your `bidirectional_map.h` file with the instructor's solution to test your implementation of these functions.

Additions/Modifications to the Core Data Structure Diagram

The `Node` objects illustrated in the diagram do not include `parent` pointers. In order to implement the forward and backward iterators you will need to add these pointers. Alternatively, you can implement your `bidirectional_map` iterators using a vector of `Node` pointers, as discussed in lecture.

Also, for extra credit, you can extend the structure to allow *many-to-one*, *one-to-many*, and *many-to-many* key/value associations. To add this feature, each node in the structure will store an STL `vector` of *link*s to `Node` pointers in the other tree rather than just a single `Node` pointer. Examples of the non-*one-to-one* interface are included in the provided code.

Performance

For this assignment we will assume that the data stored in the structure is added and removed in a mostly random fashion and that the binary tree remains sufficiently balanced so that we may claim that the maximum height of the tree structures is not significantly greater than $\log n$, where n is the number of associations stored in the structure. In lecture, we will talk about algorithms for automatically rebalancing trees (but you do not need to implement this for the homework). In your `README.txt` file include the order notation for each of the functions described above. If you complete the implementation of non-*one-to-one* associations for extra credit, you will also use k = the maximum number of links in a single `Node` in your answers.

Additional Testing & Homework Submission

We encourage you to work through the test cases in the provided `main.cpp` step-by-step, uncommenting and debugging one test at a time. It is your responsibility to add additional test cases, including examples where the template class *key* and *value* types are something other than `string` and `int`. Note: Your print function is only required to work for simple types (e.g., `int`, `double`, `char`, and short `string`s). Also, your print function is to help you debug, and does not need to exactly match the sample output. The homework submission server is configured to run your code with Valgrind to search for memory problems. Your program must be memory error free to receive full credit.

Use good coding style and comments when you design and implement your program. 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 "Academic Integrity for Homework" handout. If you did discuss the problem or error messages with anyone, please list their names in your `README.txt` file.**