Programming Project #9

Assignment Overview
In this assignment you will practice creating a custom data structure, `BiMap`, which extends the map class, without using the potential STL bases (map and set). You will create a class to do this work. It is due 11/19, Monday, before midnight on Mimir.

Background
You are going to create a container called a Bidirectional Map, `BiMap` will be used as the name of the class. A `BiMap` is a kind of associative container that has both unique keys and unique values. That is, you store values that have a key:value relationship like a map, but there is only one example of a key and one example of a value. Thus the pairing, as well as both elements, are unique. You are going to build a `BiMap` that stores string:string key:value pairs (no templating required), with underlying data structures that are organized by key and value separately for efficient access. ***You cannot use either a STL map or a set (or any other STL map/set classes)*** in the process of building this data structure, but you will use the STL vector, string, and pair classes and you can use (and moreover, are encouraged to use) any appropriate STL algorithms except sort algorithms. ***You cannot use any sort algorithm!***

Details
We provide a header file, `proj09_bimap.h`, which provides details of type for all the required methods and functions for the class `BiMap`. The basics are this. The `BiMap` class maintains two private data member `vector< pair< string, string > > ordered_by_keys_` and `vector< pair<string, string > > ordered_by_vals_`. The pairs stored in each vector are the same, but are ordered by the keys (the first element in the pair) and values (the second element in the pair), respectively. We must constantly maintain the ordering on each vector for efficient data access and manipulation. The order for both vectors is from smallest to largest as defined by the `string` STL type. We describe the methods below.

- `vector< pair <string, string> >::iterator find_key(string key)` This is a private method only usable by other `BiMap` methods (not from a main program). Uses `lower_bound`, returns an iterator to a `pair<string,string>` that is either the first pair in the vector that is equal to (by key) or greater than the key, or `ordered_by_keys_.end()` (the last two meaning that the key isn't in `ordered_by_keys_`). It must be private because `ordered_by_keys_` is private and we cannot return an iterator to a private data.

- `vector< pair <string, string> >::iterator find_value(string value)` This is a private method only usable by other `BiMap` methods (not from a main program). Uses `lower_bound`, returns an iterator to a `pair<string,string>` that is either the first pair in the vector that is equal to (by value) or greater than the value, or `ordered_by_vals_.end()` (the last two meaning that the key isn't in `ordered_by_vals_`). It must be private because `ordered_by_vals_` is private and we cannot return an iterator to a private data.

  To make `lower_bound` work, you are either going to have to write a function or a lambda that compares a `pair<string, string>` and a `string`. See `lower_bound` below.

  These functions are ***not tested*** in the Mimir test set but necessary everywhere. However, it is

essentially the use of `lower_bound`. It is a good to isolate this usage however for future projects.

- `BiMap(initializer_list< pair<string, string> > )`: Take each `pair` and place in the vectors. The `initializer_list` does not have to be in sorted order but the vector should be after you add all the elements. (Hint: write `add` first and use it here)
- `size_t size()` : returns the size of the `BiMap` (number of pairs) as an unsigned int
- `string key_from_value(string value)`: member function. Return the key associated with the value. If the value does not exist, then return the empty string.
- `string value_from_key(string key)`: member function. Return the value associated with the key. If the key does not exist, then return the empty string.
- `bool update(string, string)`: if the first string as a key is in the `BiMap`, and the second string as a value is not, update the key-value pair to the value of the second parameter. Return true. If the key is not in `BiMap`, do nothing and return false.
- `pair<string, string> remove (string element)`: member function. Look for element, as either a key or a value.
    - If found, remove both the key and value from the `BiMap` and return that `pair<key, value>` (that order).
    - if the element does not exist as either a key or a value, return a pair of empty strings.
    - if element exists as both a key and a value (possible, as unique-ness is guaranteed only among keys or only among values, for example the `BiMap` could legally contain "xx":"yy" and "yy":"xx"), remove both pairs, and return a pair with the key that was associated with element, and the value that was associated with element.
- `bool add(string,string)`: if the first string as a key or the second string as a value is in the `BiMap`, do nothing and return false. Otherwise create a pair with the argument values and insert the new pair into the vectors, in sorted order, and return true.
- `bool compare(BiMap&)` : compare the two `BiMaps` lexicographically, that is element by element using the string-key of the pairs as comparison values. If you compare two pairs, then the comparison is based on the `.first` of each pair (that is, the string-key of each pair). The first difference that occurs determines the compare result. If the calling `BiMap` is greater, return 1. If the argument `BiMap` is greater, return -1. If all of the comparable pairs are equal but one `BiMap` is bigger (has more pairs), then the longer determines the return value (1 if the first is longer, -1 if the second). If the two `BiMaps` are identical, return 0
- `BiMap merge(BiMap&)` : Merges the current `BiMap` with the `BiMap` being passed in, creating a new `BiMap` with the values of both; if both `BiMaps` have overlapping keys or values, keep those from the calling `BiMap`
- `friend ostream& operator<<(ostream&, BiMap&)`. Returns the ostream after writing the `BiMap` to the `ostream`. The formatting should have each pair colon (':') separated, and each pair comma + space separated (', '), with no trailing comma.
    - E.g., `Ann:ABCD, Bob:EFGH, Charlie:IJKL`

Requirements

We provide `proj09_bimap.h`, you submit to Mimir `proj09_bimap.cpp`

We will test your files using Mimir, as always.

Deliverables

`proj09/proj09_bimap.cpp`

1. Remember to include your section, the date, project number and comments.
2. Please be sure to use the specified directory and file name.

Assignment Notes

Look at example lower_bound.cpp in the directory

lower_bound
Your new favorite algorithm should be `lower_bound`. Look it up. It returns an iterator to the ***first*** element in a container that is "not less than" (that is, greater than or equal to) the provided search value. It requires that the container elements be in ***sorted order***, and if so does a fast search (a binary search) to find the search value. It has the following form:

```
lower_bound(container.begin(), container.end(), value_to_search_for)
```
or
```
lower_bound(container.begin(), container.end(), value_to_search_for,
binary_predicate)
```

where the `binary_predicate` takes 2 arguments: the first an element of the container and the second the `value_to_search_for`. It returns true if the element of the container is less than `value_to_search_for`.

The return value is an iterator to the either the element in the container that meets the criteria, or the value of the last element in the range searched (in this case, `container.end()`)

That means that either:

- the `value_to_search_for` is already in the container and the iterator points to it.
- `value_to_search_for` is not in the container. Not in the container means:
    - the iterator points to a value "just greater" than the `value_to_search_for`
    - the iterator points to `container.end()`

vector insert
Very conveniently, you can do an insert on a vector. You must provide an iterator and a value to insert. The insert method places the new value in front of the iterator. In collaboration with `lower_bound`, you can place an element in a vector at the location you wish, maintaining sorted order at every insert.

add
The critical method is `add`. Get that right first and then much of the rest is easy. For example, the initializer list constructor can then use `add` to put elements into the vector at the correct location (in sorted order).

sort
No use of sort allowed. If you use sort in a test case you will get 0 for that test case. Do a combination of lower_bound and vector insert to get an element where it needs to be in a vector.

Empty strings
Since empty strings are used to indicate values not found, none of the valid keys or values stored in the BiMap will be empty