# Programming Project #9

## Assignment Overview

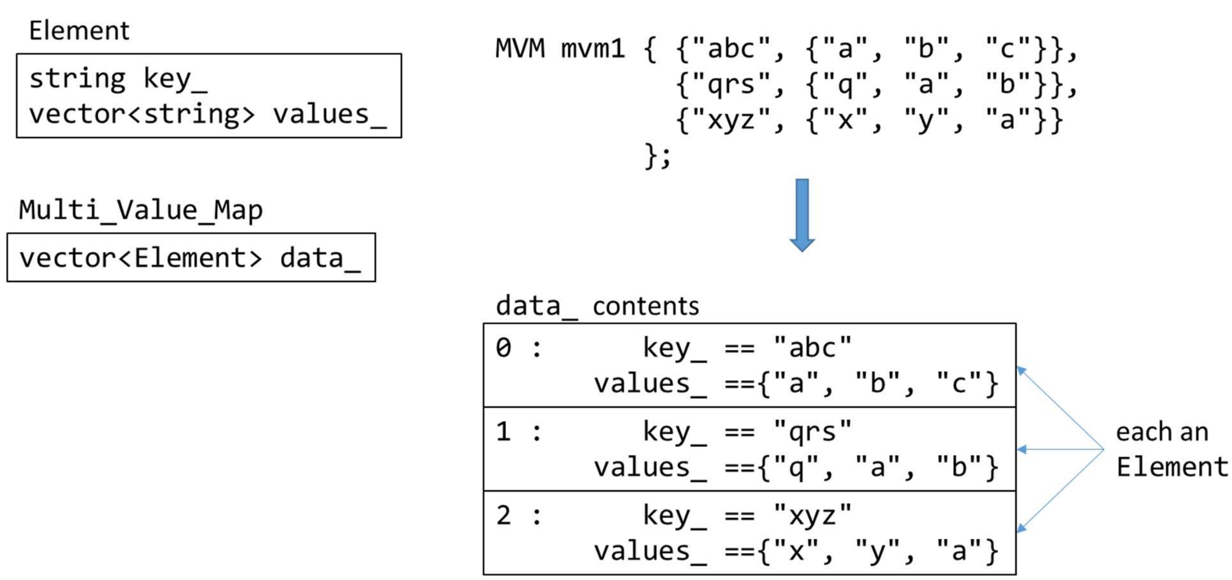
In this assignment you will practice creating a custom data structure, MVM, which extends the map class, without using the potential STL bases (map and set). You will create two classes to do this work. It is due 04/08, Monday, before midnight on Mimir. That's two weeks because of the midterm on 03/28. See the front page of the website. It is worth 60 points (6% of your overall grade).

## Background

You are going to create a container called a Multi_Value_Map but which we will shorten to just MVM A MVM is a kind of associative container that has unique keys, like a regular map, but can have associated with each key a **group** of values, not just one value. That is, you store values that have a key:{value_a, value_b, value_c, …} relationship. The MVM has the following restrictions:

- The key is unique. There is no repeat of a key in a MVM.
- The values associated with a **particular** key are also unique. A value cannot be repeated in association with a key. *However*, a value can show up associated with **different** keys (can repeat across multiple keys) and that is allowed.
- The entries in a MVM are always sorted in key order. That is, if you add a new key to a MVM, it will be placed in its proper sorted position relative to the other keys. You do not need to use the sort function to do this, see details below.
- The values associated with a key are stored in first come, first serve order. That is, the first entry in a list of values associated with a key is the first value added, the second in the list the second value added, etc.

You are going to build an MVM that stores keys as strings and values as string. No templating of your class yet. To support this work, we will also design another class called Element. You can think of Element as the payload class to be used by MVM. The organization will look something like the following:



Each `Element` has a `string key_` and a `vector<string> values_` (note the underlines trailing the data members). The `MVM` has a `vector<Element> data_` which is organized in key order. Note that the value `"a"` is repeated in multiple Element's `value_` vector but no value is repeated in the same `values_` vector. Neither is any `key_` repeated. The indicies of `data_` are shown for clarity and are not part of the actual data structure.

## Details

We provide a header file, `proj09_class.h`, which provides details of type for all the required methods and functions for the classes `Element` and `MVM`.

**`Element`**

`Element()=default`

- Default ctor. Do not need to write

`Element(string key, initializer_list<string> values)`

- Take a string key and an initializer_list values and construct an `Element` with those values.

`bool operator==(const Element&)`

- Two Elements are equal if their two keys_ are equal and if their two values_ are equal.
    - return true if the two Elements meet this condition, false otherwise.
    - this is a member function.
    - Note: you do not have to compare each of the elements in values_, just compare the vectors directly
- This will help with testing. You can see if two Elements are equal (what you think should be in the vector and what actually is). One liner, easy to write.

`friend ostream& operator<<(ostream&, Element&)`

- output the `Element` to the provided ostream (don't just print to cout, you won't pass the Mimir test).
- Look at Mimir test cases for details on output format.

**`MVM`**

`MVM()=default`

- default ctor. . Do not need to write

`MVM`(initializer_list<`Element`>)

- initialize the data_ member to the initializer_list
- is added in `initializer_list` order (see note below)

`vector<Element>::iterator find_key(string key);`

- must use the algorithm `std::lower_bound`.
- returns an iterator that points to an `Element` in data_
- return value cases are:
    - points to an `Element` in data_ which has the key
    - point to an `Element` in data_ which is just bigger than the key (thus the key isn't there).
        - if data_.end(), the key isn't there and it's bigger than all existing keys

`vector<string> find_value(string val)`

- returns a (possibly empty) `vector<string>` which is a list of all keys where `val` is located

`bool add(string key,string value)`

Should use find_keys. The cases are:

- The `key` exists. Check the `value`
    - `value` not in `values_`, push it onto the back of `values_`

    - `value` is already in `values_`, do nothing but return false
- The `key` isn't there and it is bigger than all existing keys
    - push a new `Element(key, {value})` onto the back of data_
- The `key` isn't there. The `find_key` iterator can be used to do an `insert` into data_.
- The return is always true unless the key and the value (both) already exist.

`size_t size()`
- size of `data_`

`bool remove_key(string key)`
- check if `key` is in the `MVM` (use `find_key`).
    - if yes, remove and return true
    - if not do nothing and return false

`vector<string> remove_value(string)`
- for every `Element` in the `MVM`
    - if the `value` is in the `values_` of the `Element`, remove it
    - return a `vector<string>` of all the keys where a value was removed

`friend ostream& operator<<(ostream&, MVM& )`
- print an MVM, see Mimir for format

## Requirements

We provide `proj09_class.h`, you submit to Mimir `proj09_class.cpp`

We will test your files using Mimir, as always.

## Deliverables

`proj09/proj09_class.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

**Element operator==**
You have to get this one right! Do it first. Most of the tests in Mimir use this. Nothing will work without it so check it. It isn't that hard.

**lower_bound (Look at example lower_bound.cpp in the directory)**
Your new favorite algorithm should be `lower_bound`. Look it up. It returns an iterator to the ***<u>first</u>*** `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 ***<u>sorted order</u>***, 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)
```

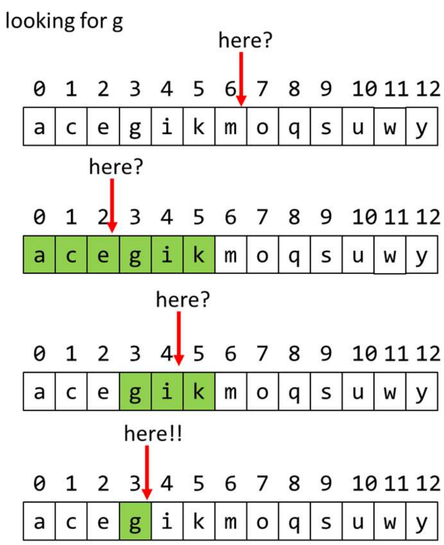
where the `binary_predicate` takes 2 arguments: the first an `Element` of the container and the second the `value_to_search_for`. It returns true if the `Element` of the container is less than `value_to_search_for`. Remember, less than of `Element` is by `key_`

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:
    - o the iterator points to a value "just greater" than the `value_to_search_for`
    - o the iterator points to `container.end()`

**Why lower_bound instead of a loop?**
Why not just use a loop to look for a key/value? Because on a sorted list `lower_bound` is very efficient. It does a binary search. If you are a Price-is-Right fan this is the search you should use in the Hi-Lo game. Look at the diagram below.




Algorithm:

- loop while not yet found (or not possible)
    - pick the middle of the current range
        - is it the value?
            - if yes, then done
            - if no
                - is it greater, look to left range
                - is it smaller, look to right range

if the elements are sorted, you can find the value quickly, or discover it is not there. This is what `lower_bound` does on a sorted list for a search. We want to be efficient so we require that:

- when you add an `Element`, you put it in the location it would go if is sorted key order (no sorting!).
- if already in sorted order, `lower_bound` is more efficient than a loop through every `Element`.

**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 <u>*in front of*</u> 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 `MVM` will be empty

**private vs. public**
You will note that all elements in the class are public. We do this to make testing easier. Any public part can be accessed in a main program which is convenient. The parts that should be private are marked. In particular `data_` and the `find_value` and `find_key` members should probably be private.

**initializer_list ctor**
It should be the case that the Elements in the `initializer_list` ctor should insert into the `MVM` in key order using `add`. However, that again makes testing harder (can't set up a simple `MVM` without getting `add` to work, and it is the most work). Thus we allow you to write the initializer_list ctor to put Elements into the `MVM` in the order of the list `Elements`. We will guarantee for our testing that anytime we use the `initializer_list` ctor we will start out with `Elements` in key order. After that maintaining that order will be up to you.