Programming Project #10

Assignment Overview

In the previous assignment you were introduced to a custom data structure, `MVM`, that did not use the potential STL bases (map and set) but used the vector class. In this project, you will re-write the `MVM` as a templated class without the use of vector by using static and dynamic arrays. It is due 04/15, Monday, before midnight on Mimir. See the front page of the website for schedule details. It is worth 60 points (6% of your overall grade).

Background

You are going to re-create the `MVM` with templating and without the use of the vector class. To do so, you will use templated dynamic and static arrays. The vector in the Element class will be replaced with a static array, and the vector in the MVM class will be replaced with a dynamic array. This means that only so many values can be associated with each key, but we can still accommodate a variable number of keys. Other than this one change, the basic properties of how the MVM class behaves remain the same. The implementation of the MVM will be different to handle the templating and lack of vectors.

Templating

Both the Element class and the MVM class will have two templates. The type of keys will be referred to as `K`, and the type of values will be referred to as `V`.

Element

Each `Element` has a `K key_` and a `V[element_array_size] values_`, as well as a new `size_t count_` to track the number of values currently being stored. The value `element_array_size` is a const global variable. Thus the size of the array in every `Element` will be the same and will be **fixed** for the duration of the program. It is not required that you grow the array, but you do have to check if it is full, preventing you from adding a new value (see the add member function).

MVM

The `MVM` has a `Element<K,V> * data_` which points to a dynamic array organized in key order, as well as a `size_t num_keys_` to keep track of the number of `Elements` presently in `data_`, and a `size_t num_elements_` to keep track of the total size of `Elements` that can be placed in the present `data_`. The `data_` array is ***dynamic*** and must grow as more Elements are added to it.

Details

We provide a header file, `proj10_class.h`, which provides details of type for all the required methods and functions for the classes `Element` and `MVM`. In order to make the process of templating easier, you will write the function definitions in the same header file .

Remember, everything is public, not because it is a good idea but because it is easier to test your code that way.

Element<K,V>

Member functions re-used from project 09:

`Element()=default`

- Default ctor. Do not need to write

`Element(K key, initializer_list<V> values)`

- Instead of copying into the vector, copy into the array `values_`. The number of elements being copied will never exceed the fixed size of `values_`. Must setup `count_` properly

`bool operator==(const Element&)`

- Behavior is identical to the previous project though, unlike vectors, you cannot compare arrays directly.

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

- Behavior is identical to the previous project

New member functions for project 10:

`bool operator<(const K&)`

- Optional; compares the key passed to an `Elements` key; done correctly, allows easier use of the `lower_bound` algorithm

MVM<K,V>

Member functions re-used from project 09:

`MVM()=default`

- default ctor. Do not need to write

`MVM(initializer_list<Element<K,V>)`

- Behavior is identical to the previous project. Must properly allocate data_, and initialize `num_keys_` and `num_elements_`

`Element* find_key(K key);`

- Behavior is identical to previous project but returns a pointer (not an iterator) to an `Element` (or one past the last `Element` if not found)

`size_t find_value(V val, K* (&keys_found))`

- We pass a `val` to search for and an array reference `keys_found` (note the odd parentheses) . The intent is to fill the array `keys_found` with all keys containing the `val`
- We assume that keys_found starts with the value nullptr. This is because find_value will create the array dynamically and fill it with the found keys.
    - If keys_found is not a nullptr, throws a runtime_error
- finds all keys where `val` is located
- creates a dynamic array pointed to by `keys_found` that holds the keys that have `val` in their `values _` array
- returns the size of the array pointed to by `keys_found`
- NOTE: `keys_found` is passed in as a parameter (as opposed to being part of the function return) to signify that the memory management is to be handled by the calling function. This is not necessarily best practices, but is intended to give you practice in managing dynamic memory

`bool add(string key, string value)`

- Behavior is identical to previous project, with the following exception:

- If you are **adding a new key to the MVM**, and the dynamic array is full (i.e. `num_keys_ == num_elements_`), you need to call the `grow()` function to make `data_` bigger.
- If you are trying to add **a new value to an Element** that has the maximum number of values (i.e. count_ == element_array_size) do not add the value and return false. Remember, the array of an `Element` is fixed size and does not change.

`size_t size()`
- Returns `num_keys_`

`bool remove_key(K key)`
- Behavior is identical to previous project

`size_t remove_value(V val, K* (&keys_found))`
- Again, `keys_found` is an array reference assumed to start as a `nullptr`. `remove_value` will create the array of keys where the value was removed dynamically and keys_found will point to that array.
    - Like before, if keys_found is not a nullptr, throws a runtime_error
- finds all keys where `val` is located, and removes `val` from the `values_` array
- creates a dynamic array pointed to by keys_found that holds the keys that had `val` in their `values _` array
- returns the size of the array pointed to by `keys_found`
- Hint: use the `find_value` function to handle most of the work
- NOTE: `keys_found` is passed in as a parameter (as opposed to being part of the function return) to signify that the memory management is to be handled by the calling function. This is not necessarily best practices, but is intended to give you practice in managing dynamic memory

`friend ostream& operator<<(ostream&, MVM& )`
- Behavior is identical to previous project

`New member functions for project 10:`

`MVM(const MVM& other)`
- Copy ctor. Constructs a new MVM with its own dynamically allocated memory that is a copy of `other`

`~MVM()`
- Destructor. Deletes any allocated memory as necessary

`void grow()`
- If the array is empty (i.e. `num_elements_` is 0 and `data_` is a `nullptr`) set the `num_elements_` to 2 and dynamically allocate `data_` to be of size 2.
- Otherwise, reallocate `data_` with twice as many num_elements_ and the correct keys stored internally, taking care to manage the dynamically allocated memory correctly

Requirements

We provide the basic `proj10_class.h`, you write the functions in the file and submit it to Mimir

We will test your files using Mimir, as always.

Deliverables

`proj10/proj10_class.h`

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. Be sure to take into account num_elements_, and num_keys_

lower_bound
Because pointers are functionally similar to iterators, we can use lower_bound in a near-identical way to the previous project

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 `Element`s 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.

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 the member variables `data_, num_keys,` and `num_elements_` and the member functions `grow, find_value,` and `find_key` 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.

main
The main program is responsible for `delete`-ing memory from either `find_value` or `remove_value`. This is not the best approach. A class should allocate and deallocate memory, thus only one element having that responsibility. Nonetheless, for simplicity we allocate in the member functions and pass that pointer to the main program. Thus the main program does the `delete []` of the memory.