Programming Project #11

Assignment Overview

In the previous assignments 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 (Project 09) or a dynamic array (Project 10). In this project, you will once again re-write the `MVM` again as a templated class using a linked list. It is due 04/22, Monday, before midnight on Mimir. See the front page of the website for schedule details. It is worth 65 points, 6.5% of your final grade.

Background

You are going to re-create the `MVM` again using a linked list for the MVM Element. The Element class will again continue using a fixed-size array for `values_` as in Project 10, but data_ in the MVM class will utilize a singly a linked list. The fixed-size of the array `values_` in `Element` means that only so many values can be associated with each key, but using a linked list we can accommodate a variable number of Elements in an MVM. Other than this one change, the basic properties of how the MVM class behaves remain the same. The implementation of the MVM will need to be updated to handle the substitution of a linked list for arrays.

As before, each `Element` has a `K key_`, a `V[element_array_size] values_`, a `size_t count_` to track the number of values currently being stored, and a new `Element* next_` to construct the linked list. The `MVM` has an `Element<K,V> * data_head_` which points to the first `Element` in the linked list as well as a `size_t num_keys_` to keep track of the number of active keys.

About Singly-Linked Lists

A singly-linked list has only the ability to move in one direction, from a location in the list towards the end, basically forward through the list. You cannot move backwards through a singly-linked list. The slides from this week describe how we could implement a doubly-linked list to overcome this limitation, but we have enough to do with singly-linked lists.

The inability to move both forwards **and** backwards means that one of the fundamental approaches we used in Project 09 and Project 10 cannot be utilized, namely the use of `lower_bound`. In fact, most of the STL algorithms will not work for this very reason. We still require that a new `Element` should be placed in the list in its sorted location when added, but we must now revert to basic linear search to find that location. That is, to find where to put the new `Element` we need to start from the beginning of the list and iterate forward, one `Element` at a time, until we come to the insertion location.

Huge Hint!

Look, there is a lot of code in the examples this week that are relevant. For the most part the examples provide most of what you need, requiring only some modifications on your part. Please feel free to use that code! I know that might make some of your nervous as you will all be working from the same code base and that you might get caught up in the plagiarism checks. We fully understand that and will take that into account.

Bottom Line: If you never looked at the examples until now (which is a real shame by the way), look at the examples from this week.

Details

We provide a header file, `proj11_class.h`, which provides details of type for all the required methods and functions for the classes `Element` and `MVM`. As in the last project, you will write the function definitions in the same header file.

`Element<K,V>`

Much of this remains the same, but we add a new data member `next_` . The data member `next_` will be a pointer to the next `Element` in the linked list. You can think of `Element` as taking the role of `Node` in the examples, where `Element` is a more complicated version of `Node`.

Functions re-used from projects 09/10:

`Element()=default`

- Default ctor. Do not need to write

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

- copy into the array `values_` and set `count_` correctly
- Sets `next_` to nullptr (no next `Element` in a list yet)

`bool operator==(const Element&)`

- Behavior is identical to the previous project. Doesn't need to account for `next_`

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

- Behavior is identical to the previous project

`bool operator<(const K&)`

- Optional; compares the key passed to the element's key

New functions for project 11:

`Element(const Element& other)`

- Copy ctor. Copies `count_` and `values_`. Sets `next_` to `nullptr.` Remember that `values_` is a fixed-size array. You know how to copy that, you just did it in project 10.

`MVM<K,V>`

Functions re-used from projects 09/10:

`MVM()=default`

- default ctor. Do not need to write

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

- Behavior is identical to the previous project. However, you are now adding Elements, in order as found in the initializer_list, into a linked list. Same idea, different data structure.

`MVM(const MVM& other)`

- Copy ctor. Creates a copy of other. The ctor for Element can copy the Element, but you need to recreate the linked list yourself. Look at the examples!

`~MVM()`

- Destructor. Deletes any allocated memory as necessary. Again, look at the examples!

`pair<Element*, Element*> find_key(K key)`

- We can no longer use `lower_bound` as we mentioned
- However, having a singly linked list means that finding only the first greater value is of limited use.

- Thus we change the return value to be an STL `pair` of pointers,
    - the `.second` points to the `Element` whose `key_` is either equal to or just greater than the parameter key
    - the `.first` points to the `Element` just behind (just previous to) the `Element` pointed to by `.second`
- This return behavior should help you to link/add and unlink/remove items from the linked list
- Given a key to search for:
    - If there is no `Element` with a smaller `.key_,` return a `nullptr` for the `.first of` the pair.
    - If there is no `Element` with a `.key_` that is greater or equal, return a `nullptr` as the `.second` of the pair.
    - If the list is empty, return a pair with both values being `nullptr`.

`MVM find_value(V val)`

- finds all keys where `val` is located
- creates an MVM where each key is a key that has `val` in its `values_`, and the only value associated with each key is `val`
- returns the new MVM

`bool add(K key,V value)`

- Behavior is identical to previous project, with the exception being that you no longer need to call grow(), just dynamically allocate a new Element as necessary

`size_t size()`

- Returns num_keys_

`bool remove_key(K key)`

- Behavior is identical to previous project

`MVM remove_value(V val)`

- finds all keys where `val` is located, and removes `val` from the `values_` array
- creates and returns a MVM of the same format used in find_value
- Hint: use the find_value function to handle most of the work

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

- Behavior is identical to previous project

`New functions for project 11:`

`MVM& operator=(const MVM& other)`

- Makes this have the same values as other
- Hint: use the copy ctor to make a copy of other that you can then use to swap data members with the local MVM

Requirements

We provide the basic `proj11_class.h`, you write the functions in the file and submit it to Mimir

We will test your files using Mimir, as always.

Deliverables

`proj11/proj11_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_

no more lower_bound
As we are using a linked list to store the MVM, we can no longer use lower_bound. However, we still require the Elements to be stored in sorted order for consistency

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 and it wouldn't work anyway as mentioned. Use `find_key` to get an `Element` where it needs to be in the linked list.

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_` and `num_keys_` and the member functions `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.