+CSE 232

Programming Project #10

Assignment Overview

In this assignment you will continue your practice in creating a custom data structure, `BiMap`. We change the specifications so that you cannot use a STL vector nor pair (as well as not use map or set or the sort algorithm). You will update your class to use **dynamic arrays** instead of STL vectors. It is due 11/28. That's a **Wednesday** because of the Thanksgiving holidays.

Background

We are going to stay with the `BiMap` problem but update its implementation. We are going to stop using vectors and pairs and do the work ourselves with underlying dynamic arrays. We will also templatize the `BiMap` to work with different types.

We do this as a better way to get familiar with the underlying problem of templated classes and dynamic arrays. The specs of the `BiMap` should be familiar to you now, so it is a matter of changing the implementation. The focus will be on this implementation.

Details

As this is a templated class you can ***only*** write a header file to solve the problem. Thus, we will provide a proto-header file, `proj10_bimap_starter.h`, which you will fill in. In the specifications below, we will focus on what is different first for this vs. the previous `BiMap`, then update the rest of the already familiar set of methods.

Class Node

Since we cannot use the STL pair, we are going to have to write our own `struct` called `Node`. This will take the place of the `pair` from the STL. Here is the header part of Node:

```
template<typename K, typename V>
   struct Node {
   K first;
   V second;
   Node() = default;
   Node(K,V);
   bool operator<(const Node&) const;
   bool operator==(const Node&) const;
   friend ostream& operator<<(ostream &out, const Node &n){
      // YOUR CODE HERE
   }
};
```

First note it is doubly templated on the Key (K) and the Value (V). Conveniently we name the data members `first` and `second` to be compatible with `std::pair`. You are required to write **in this header** file:

- 2 arg constructor
- `operator<` Compares two `Node` instances based on the first values (so that, if we need to compare using something like `lower_bound`, they would be ordered by key).
- `operator==` Similarly, two `Node` instances are equal if their two `first` values are equal (can't have duplicate keys).

- `operator<<` As discussed in the videos, and unlike the other definitions (which should appear below the `struct`), the easiest way to do `operator<<` for a templated class is to place the definition ***right here*** in the class section (not below where the other methods would go). It should print "`first:second`" to the `ostream`, whatever the values of `first` and `second` are.

Class BiMap

The underlying data member for the storage of `Node` is an array (not a vector, a basic array) of type `Node`. Since this is not a vector we have to grow the vector when we add more `Node` elements than the array can presently store. The array has to grow dynamically during the course of the run. As before, the elements of the array must be in sorted order of the Nodes, using either the key or the value to order the two arrays named `ordered_by_keys` or `ordered_by_vals`, as we did before.

As this pertains to the `BiMap` class, where we previously had the two vectors, we now have two Node arrays `ordered_by_keys_` and `ordered_by_vals_`. These are to be ordered and treated in the same way as the previous project – only now we are managing the dynamic memory ourselves. The header portion of the BiMap class should appear as follows:

```
template<typename K, typename V>
class BiMap {
private:
      Node<K,V>* ordered_by_keys_;
      Node<K,V>* ordered_by_vals_;
      size_t last_;
      size_t capacity_;
      Node<K,V>* find_key(K key);
      Node<K,V>* find_value(V value);
      void grow ();

public:
      BiMap(int sz = 2);
      BiMap(initializer_list< Node<K,V> >);
      BiMap (const BiMap&);
      BiMap operator=(BiMap);
      ~BiMap();
      size_t size();
      V value_from_key(K);
      K key_from_value(V);
      bool update(K, V);
      K remove_val(V value);
      V remove_key(K key);
      bool add(K, V);
      int compare(BiMap &);

      friend ostream& operator<<(ostream &out, const BiMap &ms){
         // YOUR CODE HERE
      }
};
```

Data Members new to Project 10

- `capacity_` – the actual size of the underlying arrays. It is the number of the `Node` elements the arrays **can hold** before they must grow.
- `last_` – the index of the first open (unusued) location in the arrays. When `last_ == capacity_` then the arrays must grow.
    - Note that these values are the same for both arrays, and will remain so, since the data contained in each is the same, only with a different ordering

Methods new to Project 10

- `1 arg constructor` (default value for size). Creates an array of size `sz` (the parameter) of type `Node` using the `new` operator. The `sz` parameter also is the is the `capacity_` of the `MapSet`. `last_` is set to 0 (the first open index).
- `grow` . This method doubles the capacity of the underlying array. It:
    - create new arrays of twice the capacity
    - copies the content of the old arrays into the new arrays
    - update `capacity_` and `ary_`, delete the old arrays
- `copy constructor`
- `operator=`
- `destructor`

  These are all methods that we have ignored previously because we took the defaults for those methods. That worked because their operations were taken care of by the underlying STL elements. Now you must provide them yourself. You have to **<u>look at the example code</u>**, videos and notes to create these. In fact, **copy the example code and modify to suit your needs here** (you won't get accused of cheating for this, I promise).

  The destructor is tested in that memory leaks are tested (Test 35). Other than that these are tested implicity in the code but YOU should test them yourself

- `K remove_val(V value)`
- `V remove_key(K key)`

  Since we no longer have the same type for the keys and values, we have to break up the remove operator into two separate functions, to either remove the value from the arrays, and return the associated key, or remove the key and return the associated value. If the key or value is not present, a default value or key should be returned, respectively

Methods modified for Project 10

- `Node<K,V>* find_key(K key)`
- `Node<K,V>* find_value(V value)`
- These both serve the same functionality as in the previous project, but return a `Node<K,V>*` instead of an iterator. You can still use `lower_bound` if you do pointer arithmetic, though the `begin()` and `end()` functions won't work as the array size is not fixed at compile time. The pointer is either the first `Node` in the array that is equal to (by key/value) or greater than the key/value, or `nullptr` (the last two meaning that the key isn't in the arrays). It must be private because the arrays are private and we cannot return a pointer to private data.

  To make `lower_bound` work, you are either going to have to write a function or a lambda that compares a `Node<K,V>` and a `K` or a `V`

  These functions are ***<u>not tested</u>*** 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< Node<K,V> > )`: Create an array of size `initializer_list.size().` Take each `Node` 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
- `K key_from_value(V value)`: member function. Return the key associated with the value. If the value does not exist, then return a default `K`
- `V value_from_key(K key)`: member function. Return the value associated with the key. If the key does not exist, then return a default `V`
- `bool update(K,V)`: if the key is in the `BiMap`, and the value is not, update the key-value pair to the parameter value. Return true. If the key is not in `BiMap`, do nothing and return false.
- `bool add(K,V)`: if the key or the value are 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. 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).
- `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 `proj10_bimap_starter.h` as a skeleton that you must fill in. You submit the completed `proj10_bimap.h` to Mimir

We will test your files using Mimir, as always.

Deliverables
`proj10/proj10_bimap.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

lower_bound
You should get how `lower_bound` works now. The difference here is now you have no iterators to work with. You must instead use pointer arithmetic.

```
lower_bound(ary_,  ary_+last_, value_to_search_for)
```

The return value is a pointer (not 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, `nullptr`)

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 `ayr_+last_`

array insert or erase
Bad news, there is no insert nor erase. You can write these separately or put them in add and remove (respectively). To make these work, you are going to have to remake the arrays.

- for insert: make a new array that has: copy of the old array up to to insert point + new element + the old array after the insert point.
    - if you made an array with new, better delete it or you will leak!
- for erase: make a new array that has: copy of old array up to the remove point, then copy after the old array to the end. In other words, skip the element being erased in the copy.

add
The critical method is `add`. Get that right first and them 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
As before, 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.