

Programming Project #9

Assignment Overview

In this assignment you will practice creating a combination data structure, MapSet, which combines a map and a set, without using either of their STL counterparts (map and set). You will create a class to do this work. It is due 04/09, Monday, before midnight on Mimir. That's two weeks because of the midterm on Wed, 3/28. Project is worth 60 pointer (6% of your overall grade).

Background

We are going to create a MapSet data structure. You cannot use either a STL map or a set in the process of building this data structure, but you may use vector, string and pair and you can use (are encouraged to use) any appropriate STL algorithms except sort. You cannot use sort! You need to keep the MapSet sorted by putting elements into the underlying vector in sorted order. Like STL sets, no two elements with the same key can exist in a set. That is, there is no duplication of a key-value pair in a MapSet

Details

We provide a header file, proj09_mapset.h, which provides details of type for all the required methods and functions for the class MapSet. The basics are this. The MapSet class maintains a private data member `vector< pair<string, long> > v_`. We interpret the pair as a key-value pair, that is the string is a key and the long is a value. That vector should always in sorted order of the string part of each pair, the key. The order is from smallest to largest as defined by the string STL type. We describe the methods below.

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

This function is 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 it however for future projects.

- `MapSet(initializer_list< pair<string, long> >)`: Take each pair and place in the vector. 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()` : size of the MapSet (number of pairs)
- `get(string)` : returns a `pair<string, long>` that is either a copy of the pair that has the string as a key or a pair with default values (that is, a `pair<string, long>` with the value `{ "", 0 }`).
- `update(string, long)` : if the string as a key is in the MapSet, update the key-value pair to the value of the long. Return true. If the key is not in MapSet, do nothing and return false.
- `remove(string)` : if the string as a key is in the MapSet, remove the associated pair and return true. If the key is not in the MapSet do nothing and return false.
- `add(string, long)` : if the string as a key is in the MapSet, do nothing and return false. Otherwise create a pair with the argument values and insert the new pair into the vector, in sorted order, and return

true.

- `compare (MapSet&)` : compare the two MapSets 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 MapSet is greater, return 1. If the argument MapSet is greater, return -1. If all of the comparable pairs are equal but one MapSet is bigger (has more pairs), then the longer determines the return value (1 if the first is longer, -1 if the second).
- `mapset_union (MapSet&)` . Return a new MapSet that is a union of the two MapSets being called. Again, comparison on whether an element is in the MapSet is based on the key. There is an order here. If the two MapSets have the same key but different values, then the key-value of the calling MapSet is the one that is used.
- `mapset_intersection (MapSet&)` . Return a new MapSet that is the intersection of the two MapSets being called. Again, comparison on whether an element is in the MapSet is based on the key. There is an order here. If the two MapSets have the same key but different values, then the key-value of the calling MapSet is the one that is used.
- `friend ostream& operator<< (ostream&, MapSet&)` . Returns the ostream after writing the MapSet to the ostream. The formatting should have each pair colon (':') separated, and each pair comma + space separated (', '). E.g., Ann:1234, Bob:3456, Charlie:5678

Requirements

We provide `proj09_mapset.h`, you submit to Mimir `proj09_mapset.cpp`

We will test your files using Mimir, as always.

Deliverables

`proj09/proj09_mapset.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).

set_union and set_intersection

These are probably not as helpful as you might think. See the [setops video from week 8](#). You can have repeated elements using these algorithms on a vector, [which we do not want](#). You probably have to write it yourself. Think about it, draw some pictures, see if you can figure it out.

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.