CS 540 Programming Assignment 2: Container

Due March 16th, 11:59 PM.

(This document was last modified on Wednesday, March 18, 2015 at 04:19:24 PM.)

`Map` Class Template

Implement a container class template named `Map` similar to the `std::map` class from the C++ Standard Library. Such containers map a key to some other object, which we will call the *mapped* object. Note that C++ terminology uses *object* object even for fundamental types such as `int`'s. (The mapped value is sometimes just called the value, but we will avoid this terminology so as to be consistent with the Standard Library.) Your `Map` class template will have two type parameters, *Key_T* and *Mapped_T*, denoting the key type and the mapped type, respectively. Note that, as in `std::map`, the mapped type values themselves must be in your map, not pointers to the values.

You may assume that the key types and mapped types are copy constructible, move constructible, and destructible. You may assume that key types have a less-than operator (<), and an equality operator (==), as free-standing functions (not member functions). You may also assume mapped types have an equality comparison (==). You may not assume that either class has default constructor or an assignment operator. You may only assume that a mapped type that is used with `operator[]` may be default initialized

You may *not* make any other assumptions. (Check with us if there is any doubt.)

Your `Map` class must expose three nested classes: `Iterator`, `ConstIterator`, and `ReverseIterator`. None of these classes should permit default construction.

An iterator is an object that points to an element in a sequence. The iterators must traverse the `Map` by walking through the keys in sorted order. Iterators must remain valid as long as the element they are pointing to has not been erased. Any function that results in the removal of an element from a map, such as `remove`, will invalidate any iterator that points to that element, but not any other iterator.

Your map implementation must be completely contained in your `Map.hpp` file. I do not believe that you will need a `Map.cpp` file, but you may have one if you wish.

Additionally, your class must meet the following time complexity requirements: `O(lg(`N`))` average-case for key lookup and insertion, and `O(1)` for all iterator increments and decrements. You may use a binary search tree or a skip list. Because the time complexity requirement is an average case requirement, you do not need to balance your BST.

All classes should be in the `cs540` namespace. Your code must work with test classes that are not in the `cs540` namespace, however. Your code should not have any memory errors or leaks as reported by `valgrind`. Your code should compile and run on the `remote.cs.binghamton.edu` cluster. Your code should not have any hard-coded, arbitrary limits or assumptions about maximum number of elements, maximum sizes, etc.

Preliminary test code is here. It's still in progress, so report any mismatches/problems.

- Test 1
- Test 2
- Minimal
- Morse Code Example

Template

Declaration	Description
`template <typename` *Key_T*`, typename` *Mapped_T*`> class Map;`	This declares a `Map` class that maps from *Key_T* objects to *Mapped_T* objects.

Type Member

Member	Description
`ValueType`	The type of the elements: `std::pair<const` *Key_T*`,` *Mapped_T*`>`.

Public Member Functions and Comparison Operators of Map

Prototype	Description
Constructors and Assignment Operator	
`Map();`	This constructor creates an empty map.
`Map(const Map &);`	Copy constructor.
`Map(Map &&);`	Move constructor. Must not duplicate any existing entries.
`Map &operator=(const Map &);`	Copy assignment operator. Value semantics must be used. You must be abl to handle self-assignment.
`Map& operator=( Map&&);`	Move assignment. Must not duplicate any existing entries.
`Map(std::initializer_list<std::pair<const` *Key_T*`,` *Mapped_T*`>>);`	Initializer list constructor. Support for creation of `Map` with initial values.

	ex: `Map<string,int> m{{"key1", 1}, {"key2", 2}};`
`~Map();`	Destructor, release any acquired resources.
Size	
`size_t size() const;`	Returns the number of elements in the map.
`bool empty() const;`	Returns `true` if the Map has no entries in it, `false` otherwise.
Iterators	
`Iterator begin();`	Returns an `Iterator` pointing to the first element, in order.
`Iterator end();`	Returns an `Iterator` pointing one past the last element, in order.
`ConstIterator begin() const;`	Returns a `ConstIterator` pointing to the first element, in order.
`ConstIterator end() const;`	Returns a `ConstIterator` pointing one past the last element, in order.
`ReverseIterator rbegin()`	Returns an `ReverseIterator` to the first element in reverse order, which is the last element in normal order.
`ReverseIterator rend()`	Returns an `ReverseIterator` pointing to one past the last element in reverse order, which is one before the first element in normal order.
Element Access	
`Iterator find(const` *Key_T* `&);` `ConstIterator find(const` *Key_T* `&) const;`	Returns an iterator to the given key. If the key is not found, these functions return the `end()` iterator.
Mapped_T `&at(const` *Key_T* `&);`	Returns a reference to the mapped object at the specified key. If the key is not in the Map, throws `std::out_of_range`.
`const` *Mapped_T* `&at(const` *Key_T* `&) const;`	Returns a const reference to the mapped object at the specified key. If the key is not in the map, throws `std::out_of_range`.
Mapped_T `&operator[](const` *Key_T* `&);`	If key is in the map, return a reference to the corresponding mapped objec If it is not, default constructs a mapped object for that key and returns a reference to it. This operator may not be used for a *Mapped_T* type that does not support default construction.
Modifiers	

`std::pair<Iterator, bool> insert(const ValueType &);`
 	
`template <typename` *IT_T*`>`
`void insert(`*IT_T range_beg*`,` *IT_T range_end*`);`
	The first version inserts the given pair into the map. If the key does not already exist in the map, it returns an iterator pointing to the new element and `true`. If the key already exists, it returns an iterator pointing to the element with the same key, and `false`.
	
	The second version inserts the given object or range of objects into the map. In the second version, the range of objects inserted includes the objec *range_beg* points to, but not the object that *range_end* points to. In other words, the range is *half-open*. The iterator returned in the first version points to the newly inserted element. There must be only one constructor invocation per object inserted. Note that the range may be in a different container type, as long as the iterator is compatible. For examaple, it might be from a `std::vector`. Therefore, the range insert is a member template.

`Iterator insert(std::pair<const Key_T,` *Mapped_T*`>&&);`	Inserts by moving, rather than copying, the provided key-mapped pair.
`void erase(Iterator` *pos*`);`	Removes the given object from the map.
`void remove(const` *Key_T* `&);`	Removes the element that is equal to the provided key. Throws `std::out_of_range` if the key is not in the `Map`
`void clear();`	Removes all elements from the map.
Comparison	

`bool operator==(const Map &, const Map &);`
`bool operator!=(const Map &, const Map &);`
`bool operator<(const Map &, const Map &);`
	These operators may be implemented as member functions or free functions, though implementing as free functions is recommended. The first operator compares the given maps for equality. Two maps compare equal if they have the same number of elements, and if all elements compare equal. The second operator compares the given maps for inequality. You may implement this simply as the logical complement of th equality operator. For the third operator, you must use lexicographic sorting. Corresponding elements from each maps must be compared one-by-one. A map M_1 is less than a map M_2 if there is an element in M_1 that is less than the corresponding element in the same position in maps M_2, or if all corresponding elements in both maps are equal and M_1 is shorter than

	M_2. Map elements are of type `ValueType`, so this actually compares the pairs.

Public Member Functions of `Iterator`

Prototype	Description
Map<*Key_T, Mapped_T*>::Iterator	
`Iterator(const Iterator &);`	Your class must have a copy constructor, but you do not need to define this if the implicit one works for your implementation. (Which is what I expect in most cases.)
`~Iterator();`	Destructor (implicit definition is likely good enough).
`Iterator& operator=(const Iterator &);`	Your class must have an assignment operator, but you do not need to define this if the implicit one works for your implementation. (Which is what I expect in most cases.)
`Iterator &operator++();`	Increments the iterator one element, and returns a reference to the incremented iterator (preincrement). If the iterator is pointing to the end of the list, the behavior is undefined.
`Iterator operator++(int);`	Increments the iterator one element, and returns an iterator pointing to the element prior to incrementing the iterator (postincrement). If the iterator is pointing to the end of the list, the behavior is undefined.
`Iterator &operator--();`	Decrements the iterator one element, and returns a reference to the decremented iterator (predecrement). If the iterator is pointing to the beginning of the list, the behavior is undefined. If the iterator has the special value returned by the `end()` function, then the iterator must point to the last element after this function.
`Iterator operator--(int);`	Decrements the iterator one element, and returns an iterator pointing to the element prior to decrementing (postdecrement). If the iterator is pointing to the beginning of the list, the behavior is undefined. If the iterator has the special value returned by the `end()` function, then the iterator must point to the last element after this function.
`ValueType &operator*() const;`	Returns a reference to the `ValueType` object contained in this element of the list. If the iterator is pointing to the end of the list, the behavior is undefined. This can be used to change the *Mapped_T* member of the element.
`ValueType *operator->() const;`	Special member access operator for the element. If the iterator is pointing to the end of the list, the behavior is undefined. This can be used to change the *Mapped_T* member of the element.

Public Member Functions of ConstIterator

This class has all the same functions and operators as the `Iterator` class, except that the dereference operator (`*`) and the class member access operator (`->`), better known as the arrow operator, return const references.

You should try to move as many of the operations below as possible into a base class that is common to the other iterator types.

Prototype	Description
Map<*Key_T, Mapped_T*>::ConstIterator	
`ConstIterator(const ConstIterator &);`	Your class must have a copy constructor, but you do not need to define this if the implicit one works for your implementation. (Which is what I expect in most cases.)
`ConstIterator(const Iterator &);`	This is a conversion operator.
`~ConstIterator();`	Destructor (implicit definition is likely good enough).
`ConstIterator& operator=(const ConstIterator &);`	Your class must have an assignment operator, but you do not need to define this if the implicit one works for your implementation. (Which is what I expect in most cases.)
`ConstIterator &operator++();`	Increments the iterator one element, and returns a reference to the incremented iterator (preincrement). If the iterator is pointing to the end of the list, the behavior is undefined.
`ConstIterator operator++(int);`	Increments the iterator one element, and returns an iterator pointing to the element prior to incrementing the iterator (postincrement). If the iterator is pointing to the end of the list, the behavior is undefined.
`ConstIterator &operator--();`	Decrements the iterator one element, and returns a reference to the decremented iterator (predecrement). If the iterator is pointing to the beginning of the list, the behavior is undefined. if the iterator has the special value returned by the `end()` function, then the iterator must point to the last element after this function.
`ConstIterator operator--(int);`	Decrements the iterator one element, and returns an iterator pointing to the element prior to decrementing (postdecrement). If the iterator is pointing to the beginning of the list, the behavior is undefined. if the iterator has the special value returned by the `end()` function, then the iterator must point to the last element after this function.
`const ValueType &operator*() const;`	Returns a reference to the current element of the iterator. If the iterator is pointing to the end of the list, the behavior is undefined.
`const ValueType *operator->() const;`	Special member access operator for the element. If the iterator is pointing to the end of

	the list, the behavior is undefined.

Public Member Functions of `ReverseIterator`

Prototype	Description
Map<*Key_T, Mapped_T*>::ReverseIterator	
`ReverseIterator(const ReverseIterator &);`	Your class must have a copy constructor, but you do not need to define this if the implicit one works for your implementation. (Which is what I expect in most cases.)
`~ReverseIterator();`	Destructor (implicit definition is likely good enough).
`ReverseIterator& operator=(const ReverseIterator &);`	Your class must have an assignment operator, but you do not need to define this if the implicit one works for your implementation. (Which is what I expect in most cases.)
`ReverseIterator &operator++();`	Increments the iterator one element, and returns a reference to the incremented iterator (preincrement). If the iterator is pointing to the end of the list, the behavior is undefined.
`ReverseIterator operator++(int);`	Increments the iterator one element, and returns an iterator pointing to the element prior to incrementing the iterator (postincrement). If the iterator is pointing to the end of the list, the behavior is undefined.
`ReverseIterator &operator--()`	Decrements the iterator one element, and returns a reference to the decremented iterator (predecrement). If the iterator is pointing to the beginning of the list, the behavior is undefined. If the iterator has the special value returned by the `end()` function, then the iterator must point to the last element after this function.
`ReverseIterator operator--(int)`	Decrements the iterator one element, and returns an iterator pointing to the element prior to decrementing (postdecrement). If the iterator is pointing to the beginning of the list, the behavior is undefined. If the iterator has the special value returned by the `end()` function, then the iterator must point to the last element after this function.
`ValueType &operator*() const;`	Returns a reference to the `ValueType` object contained in this element of the list. If the iterator is pointing to the end of the list, the behavior is undefined. This can be used to change the *Mapped_T* member of the element.
`ValueType *operator->() const;`	Special member access operator for the element. If the iterator is pointing to the end of the list, the behavior is undefined. This can be used to change the *Mapped_T* member of the element.

Comparison Operators for Iterators

These operators implemented as member functions or free functions. I suggest that you use free functions, however.

Member	Description

`bool operator==(const Iterator &, const Iterator &)`
`bool operator==(const ConstIterator &, const ConstIterator &)`
`bool operator==(const Iterator &, const ConstIterator &)`
`bool operator==(const ConstIterator &, const Iterator &)`
`bool operator!=(const Iterator &, const Iterator &)`
`bool operator!=(const ConstIterator &, const ConstIterator &)`
`bool operator!=const Iterator &, const ConstIterator &)`
`bool operator!=const ConstIterator &, const Iterator &)`
	You must be able to compare any combination of `Iterator` and `ConstIterator`. Two iterators compare equal if they point to the same element in the list. Two iterators may compare unequal even if the *T* objects that they contain compare equal. It's not strictly necessary that you implement the above exactly as written, only that you must be able to compare the above. For example, if your `Iterator` inherits from `ConstIterator`, then you may be able to get some of the above comparisons autumatically via implicit upcasts.

`bool operator==(const ReverseIterator &, const ReverseIterator &)` `bool operator!=(const ReverseIterator &, const ReverseIterator &)`	It's not strictly necessary that you implement the above exactly as written, only that you must be able to compare the above. For example, if your `ReverseIterator` inherits from `Iterator`, then you may be able to get some of the above comparisons autumatically via implicit upcasts.

Submission

You must submit a compressed (gzip) tar file named `cs540p2_`*lastname_firstname*`.tar.gz` containing the files through BlackBoard. The tar file should expand to the single directory `cs540p2_`*lastname_firstname* with the files in it. Of course, in all cases *lastname* and *firstname* should be replaced with your actual last name and first name. So, in my case, I would submit a tar file named `cs550p1_chiu_kenneth.tar.gz` that expands to the directory `cs550p1_chiu_kenneth` with my files in it. The files you should submit include `Map.hpp` and a `README`. If your code can directly be compiled with the provided test code, and passes all tests, then you don't need to submit anything else. Otherwise, you must submit your own test code, makefile, and anything else needed to compile and run your test code. In your `README` you must explain what you can and cannot do. You will only get credit for functionality that is tested in your test code. This tar file must **not** contain any `.o` files, executables, or core files.