CSE250, Fall 2009 **Project 1** **Due Fri. 10/30, 11:59pm**

A Vector–Linked List Tandem With Iterator

`online submission only---no hardcopy`

Reading

For Monday, read this project description, (re-)read the parts of the text it cites, (re-)read Section 7.3 on binary search, and also read Chapter 5, skim/skipping the "Case Studies" on pp320–324 and 333–350. For later in the week, read Chapter 6, skim/skipping pp381–397. (The other case studies, on palindromes and queue-maintenance, are good to read.)

Short Task Statement and Motivation

Program a templated data structure class `Valli<I>` that combines a **v**ector **a**nd **l**inked-**l**ist with **i**terator. The argument type for `I` may be assumed to already implement `operator<` and the list must be kept sorted by that operation (see text, pp285–291). (You may also assume `I` has a public zero-parameter constructor and implements a public `str()` method to write items as strings.) The `Valli` constructor takes an integer argument `r` (or call it `ratio`) and builds a `vector` whose *intent* is to provide a direct link to every `r`-th cell of the list. This allows a standard `insert` or `find` method to do a binary search over the array to find the direct link closest to where the new item should go (or to where a matching item may be found), and then take at most r more steps down the list, for a running time of $O(\log(n/r) + r)$ that vastly improves on the "$O(n)$" for linked lists alone. Moreover, inserts and removals can be done in a further $O(1)$ time without the "$O(n)$" overhead for splicing a vector. *However*, many inserts in the same spot will degrade the "`r` steps" guarantee, and so the class requires a sporadic `refresh` of the direct links to be every `r`-th cell again, adding links to the vector as needed. You should also program a `str()` method to facilitate outputting the linked list's items.

The nested `Valli<I>::iterator` class only needs to provide the functionality of a `forward_iterator`, and so the list is allowed to be only singly linked (though double-links may make your code easier to write). In addition to the above-mentioned constructor you must provide a destructor `~Valli<I>()` that deletes the cells, but you are allowed to *disable* the copy-constructor and `operator=` by declaring them `private` with empty bodies. The sortedness requirement implies that you may not assign an arbitrary item to an iterator location, and also limits your options on insertion. In all other respects the spec parallels the text's "Programming Project 2." on page 309, which references its Project 1. A full checklist of the public interface this implies is below. No client task is specified, and there is no separate project report—though logic-comments are expected at important junctures and are part of the grading scheme as on Assignment 3, and various testing clients *for you to adapt* will be provided.

What to Submit

Because this is a template class, there will not be a separate `.cpp` file. We disagree with the text's practice of putting the nested iterator class in a separate `.h` file and `#include`-ing it in the middle of the code, and we "punt" on the text's practice (not mentioned until p328 in Ch. 5) of putting template bodies in a separate "`.tc`" file. Thus we expect all of your implementation code to be in one file, `ValliNNN.h` where NNN are your initials as before. You may write all bodies inside the class braces (a-la Java), or put them outside the braces but in the same file (giving the exact same syntax and result as if you did the "`.tc`" thing). You must still call the class just `Valli` with no initials—unlike in Java this need not match the file name.

Submit `ValliNNN.h`, *and* submit a makefile, client file with `main`, and a `.h`, `.cpp` pair for a template argument class that implements the $<$ operator. This is the same number and kind of files referenced by `LinkMain.make`, which you are welcome to modify (with `-Wall` enabled).

Your argument class and client need not be fully or even halfway original—as a bargain-basement option you can just add an `operator<` to the `StringWrap.{h,cpp}` files, coded outside the class. However you may not use just `string` (as in `Valli<string>`) or any other predefined C++ type—it must be a used-defined class.

Rules and Grading

You are welcome to adapt code from the text, particularly the iterator code on pp279–281 and some of the methods before and after it. (See also "Tips" below.) You are also welcome to rummage around in `~regan/cse250/PROJECTS/S00BASE/` for code blocks to examine and/or adapt. (Most notably its `ISR` sub-folder has a singly-linked list class `SList` with a somewhat-different interface based on an older text, and in particular you may adapt its `toString()` body for your `str()` method even if you use a doubly-linked list.) Having said that, *individual-work rules of academic integrity are in effect for your* `Valli` *code*. This is not a group-work assignment, except as may be provided in recitations only. You may discuss among you templates and iterator syntax, vis-a-vis Chapter 4 (plus 5–6) and lecture notes, and (!) may share argument classes and testing clients, but you may not share code bodies in `Valli`. Put another way, treat `Valli` as *black-box testing* (p149). Logic comments you write must be your own. The "Minimal Coding Guidelines" on the course webpage are now mandated, and you may not put `using namespace std;` in any file other than the one with `main`, *after* all includes—in other `.h` and `.cpp` files you must have separate `using` statements like `using std::vector;` for any items you wish to use without the `std::` prefix. Throwing exceptions as the text shows is optional, and in any event you need not `catch` them (the text doesn't).

Grading is 120 pts. for `Valli.h`, 18 for logic comments, and 12 for the makefile, argument class, and client with some (additional) original lines by which you've tested your code (on `timberlake`, not just trusting another's client and your home system), for 150 pts. total.

The STL-based Public Interface

Except for the constructors and `refresh`, the following names and type signatures conform to the ANSI C++ Standard Template Library, specifically for the `multiset` container—as shown in the "Member map" table at `http://cplusplus.com/reference/stl`. The "multi" means that we do not forbid storing duplicate items—which in particular simplifies the return type of `insert` compared to the ones for `set` or `map`. The main difference from the text is that `insert` standardly returns an iterator that fingers the just-inserted item, rather than be `void`. Otherwise this parallels what the text indicates in Chapter 4, Project 2. Note that the text's `Item_Type` can be simply `I` in your code, and we are not making the methods or even the destructor `virtual`.

In `Valli<I>`:

Name and Signature	*Required Behavior*
`explicit Valli<I>(size_t ratio)`	Build empty container, save ratio.
`Valli<I>()`	Default constructor, you choose default ratio.
`~Valli<I>()`	Destructor. (OK to omit `<I>` within class.)
`iterator insert(const I& item)`	Add item, preserving sortedness. Also return an iterator on the just-inserted item.
`iterator find(const I& item) const`	Return iterator on item, `end()` if not found.
`void erase(iterator itr)`	Remove cell fingered by `itr`.
`iterator begin() [const?]`	Return iterator on first (least) item.
`iterator end() [const?]`	Return iterator one place past last item.
`size_t size() const`	Number of stored elements.
`bool empty() const`	Equivalent to `size() == 0`, and to `begin() == end()`.

Extra, non-STL methods:

`void refresh(size_t newRatio = this->ratio)`	
`string str() const`	assume item.str(), separate items by newlines.

In `Valli<I>::iterator`: (which == just `iterator` inside class braces)

Name and Signature	*Required Behavior*
[private constructor(s)!—client must ask container to generate one]	
[destructor?]	good design may not need one.
[`iterator(const iterator& other)`?	ditto.
`iterator& operator=(const iterator& rhs)`?	ditto.
`I& operator*() const` [see Note]	Return item the iterator is on.
`iterator& operator++()`	Advance, return new self.
`iterator operator++(int)`	Advance, return copy of old self.
`bool operator==(const iterator& rhs) [const?]`	true iff on same cell.
`bool operator!=(const iterator& rhs) [const?]`	(this must be coded too!)

You are not required to implement all the iterator functionality the text shows for a sorted list on pages 279–281; it is OK to skip decrement and `operator->`. You are not required to provide a `const_iterator` either. The STL allows `begin()` and `end()` to be non-`const` with a regular `iterator` for technical reasons not in our scope. It is OK for you to make them `const` because there is a general principle that *appending* `const` *to a method prototype never makes the method less usable!* (Adding `const` to a return type is a different story. . . and of course you can't declare a method `const` if it does modify a top-level field (unless that field is excepted as `mutable`).) To avoid a "template friend gotcha" we are following the text by coding `operator==` and `operator!=` inside the class as members, but then there is no standard to mandate that these members be `const`, and the text omits that on pp280–281. On the same principle, you *should* make your `begin()`, `end()`, `operator==`, and `operator!=` all `const`. [Note: The `I&` return of `operator*` is hence a "hole," whose standard fix is to make a `const_iterator` returning `const I&`, but per above we're postponing that, and rather than return `I` by value, we stick with `I& operator*` for now.]

As on pp279–281 you will need some `friend`-ing and field(s) in your nested `iterator` class by which it can access the current cell it's attached to. You may also need to keep a reference to the vector or the head node or to the enclosing `Valli<I>` object as a whole. One important difference from Java is that a C++ nested class cannot see members of the enclosing class directly—the technical term for this is that C++ has only what Java calls `static` inner classes. You will also need to write one-or-more iterator constructors that are kept `private`. If you use singly-linked cells then you will likely need to require a pointer to the cell *before* the one the iterator is formally "on," adding a dummy cell above the first item for this purpose. Any cell class or other class need not be nested but goes in the same file.

You need not provide the multiple versions of the constructors and several methods that the STL actually gives. You may add private fields and methods at-will, and one private method in `Valli` is tantamount to being required:

`void insert(const iterator& pos, const I& item)`: insert `item` *before* the cell that the iterator `pos` is formally "on"; append item to the end if `pos == end()`.

This is exactly what the `insert` at the bottom of page 308 becomes when you do the change in "Programming Project 2." Why is it `private`? This is because of the sortedness requirement. Unrestricted `public` use of this method would allow users to break that class invariant, and then the speedup of binary search would be kissed goodbye. Doing this `insert` as a private

overloaded method will help break up your public `insert` method body into manageable pieces. The same may be said of the binary-search routine, which will be used by your public `find` as well. Finally, disabling the `Valli` copy-constructor and `operator=` (by declaring them private with empty bodies) is not only well-motivated but saves work!

Implementation Options and Tips

As mentioned above, you may choose to make a singly or doubly linked list. The latter involves managing an extra `prev` field but makes it easier not to paint oneself into a corner with pointers. The iterator for a singly-linked list needs to keep track of the cell *before* the cell you mean it to be "on." With either option you may choose to have a "dummy cell" at the end and/or start—then the standard assumption that the argument for `I` has a zero-parameter constructor comes into play. An adventurous third option is to try to "wrap" `std::list` as the text does for its own `Ordered_List` class.

The other main option concerns the vector of "milepost" entries. The simplest and naturally-expected idea is to make it a `vector<Cell*>` of ordinary pointers to the cells. The alternative, once you've coded the `iterator` class, is to make it a vector of iterators, i.e. "smart pointers." The latter would feel like a savings especially if you've used a singly-linked list, as your iterator class will already have dealt with the "cell before" issue, and you wouldn't need to wrestle with it directly again while coding. Either way, a key issue is to avoid the `erase` method munging one of the "mileposts" when deleting a cell. You may move the milepost forward or backward—but if two mileposts wind up on the same cell, that's a time to call `refresh`. Coding your binary search as a private method that returns the milepost before the place you're looking for is therefore recommended.

The final issue is how to manage the milepost vector and when else to call `refresh`. Initially for an empty container it should have just the two entries standing for head and tail—or for begin and end if you use iterators—and one or both may be null or point to a dummy cell. Keep track of the number n of entries as a private field—then your `size()` method can simply return it. Given an initial value of r, you should wait until $r = n + 1$ before calling `refresh` in a way that will automatically add a third middle entry. (Alternatively, you may wait until $n = 2r$ so that it goes exactly in the middle.) Note that you can call `clear()` on the vector and `push_back` entries to rebuild it—then you will want to maintain a separate `head` field in the `Valli` class for the first linked-list node after all (and if you have a "dummy head" you'll have the issue of whether the first vector entry should go to it or to the first "real" cell). Theoretically r should grow as $\log n$ in order to "balance" the $O(\log n)$ time for binary search and the extra up-to-r steps for subsequently finding the exact cell needed. However, if you choose an initial value like $r = 20$—as suggested by the tradeoff points shown in class for "simple" versus "asymptotically best" code—that's good for n up to a million!

Relevance: This is the simplest data structure I know that does any one of the following three, let alone all three: (a) covers the important points in Chapter 4 and Chapter 7 (section 7.3), (b) competes with the advanced standard containers in all operations, and (c) still(?) doesn't have C++ answer code readily available on the Internet. Regarding (c), if you find some please let me know privately, and I'll adjust accordingly. Based on my experience, however, such code as there is would be more work to adapt than the code already in our textbook—which you are expressly welcome to use sans-citation. *You may not publicly post your answers.* Some motivation for competing with the standard containers is given in the article "Why you shouldn't use `set`..." by Matt Austern (`http://lafstern.org/matt/col1.pdf`), which describes binary search on a sorted vector without the list and mileposts.