ECE498SL (409 pending): Engineering Parallel Software **Spring 2012**
Lab #1 **Due: Thursday 5 April**

Developing a C++ Template Class

Your task in this assignment is to create a C++ template implementation of a skip list, a randomized data structure that gives good expected performance for insertions, deletions, and lookups of elements. The objectives in this assignment are to give you some direct experience in developing a template, to expose you to a simple randomized data structure, and to give you an opportunity to do some measurement and performance tuning.

Since this assignment is a lab, you must do it on your own.

My implementation is under 300 lines, including lots of comments, so the main difficulties will be in understanding the data structure and in learning to develop a template. Both of these topics may be discussed amongst yourselves or with me, but do not exchange code.

To maximize your learning, I discourage you from reading the skip list implementations available online until after you have tried to develop your own, and of course you may not make use of others' code directly, regardless of the source.

Rather than try to describe the ideas behind a skip list in this document, I refer you to the descriptions online. Bill Pugh's original paper (Communications of the ACM, 33(6):668-676) should be available to you from the library using any Illinois machine. The Wikipedia description is another starting point.

The Interface

Here are a few specifics about how your template must work and the functions that it must provide.

You must define in `SkipList.h` (the only file that you may modify) a template class SkipList taking two class parameters. The first class parameter—call it Element—is the type of object to be contained in the SkipList. The second class parameter—call it Comparison—must refer to a class that provides a way to compare two Elements. Any class to be passed as a SkipList's Comparison class must define

```
bool operator() (const Element& elt1, const Element& elt2) const;
```

The SkipList code should create one instance of the Comparison class for each SkipList and make use of this Comparison instance to compare elements as necessary. Your template must also provide a default class (probably also templated based on Element) that defines the `operator()` as above to make use of `operator<` with the same signature (two Elements in, and a `bool` in return).

Iterators: You must also define a helper class called `iterator` within your SkipList definition. The `iterator` class is used to refer to elements within a skip list, which is necessary when walking over all elements or simply finding a single element in the list.

For the `iterator` class, you must define a dereference operator (`operator*`) that should return a constant Element reference to avoid requiring that you deal with the issues involved if a user wants to change an element inside a SkipList in place. If you *want* to handle such changes, I suggest adding a helper class that removes/reinserts or casts as appropriate to use, as we did with array notation in an earlier assignment.

You must also define both forms of `operator++`, which should return `iterator` references.

Finally, you must define equality and inequality comparisons (`operator==` and `operator!=`) on the class `iterator`. These should both return Boolean values.

Simple Functions: The simpler functions to write include a default constructor (no arguments), a destructor—be sure that you do not leak memory, functions for walking over an entire SkipList, and a debugging function for printing the contents of a SkipList.

To enable looping over a SkipList, define `begin` and `end` functions that each return an `iterator`. Note that `end` should not refer to a valid element, but should 'point' to the element after the end of the SkipList. Note also that these functions cannot be treated as constant on the SkipList, since the iterators returned must be usable for erasing elements.

If you want to create a `const_iterator` type as well, feel free, but I feel that the added learning is minimal. You might benefit from creating forward and reverse iterators in both constant and non-constant forms and trying to unify the code for these four combinations, but that effort is beyond the scope of this assignment.

The debugging function, `void printLevels ()`, should print all levels of your SkipList by using `cout << Element&`. The specific format of the output is otherwise up to you.

Main Operations: The main functions for the SkipList include insertion (`insert`), lookup (`find`), and deletion (`erase`).

The `insert` interface takes a constant Element reference and returns a Boolean. If the Element is already in the table (use the Comparison class in both directions to determine equality), the function does nothing and returns `false`. Otherwise, insert the new element and return `true`. Note that your SkipList **must make a single private copy** of the Element referenced by the argument.

The `find` interface also takes a single constant Element reference, but returns an iterator. If the Element is present in the SkipList (again, use the Comparison class in both directions to determine equality), the iterator returned should refer to the matching element in the SkipList. If no such element is present, the iterator returned should match that returned by the function `end`.

Finally, there are two interfaces for `erase`. Both return a Boolean: `true` if an element was deleted from the SkipList, or `false` otherwise. One interface takes a single constant Element reference and searches for a matching element. The second interface takes a single constant iterator reference and erases the element referred to by the iterator. If an iterator matching `end` is passed to `erase`, the function should simply return `false`.

In summary, the SkipList interface functions are as follows:

```
SkipList ();
~SkipList ();
iterator begin ();
iterator end ();
void printLevels () const;
bool insert (const Element& newElt);
iterator find (const Element& tgt);
bool erase (const iterator& it);
bool erase (const Element& tgt);
```

Randomization: A comment is in order in regard to the randomization of your SkipList. In a real implementation, either the template should include another helper class that provides a private, static seed for one of the standard library's randomization functions (and initializes the seed randomly, possibly with some kind of deterministic option as well) or uses another default class argument to allow a user to specify their own random number routine.

In your implementation, you should simply use the `rand` library call in the standard library. The programs that I've given you initialize the generator for this call.

Testing

I have provided nine programs (using seven source files) that you must get to compile with neither warnings nor errors. Some of these programs simply test the functionality of your SkipList template.

A few of the specifics tested include...

- Does `begin` work on an empty SkipList?
- Can one create a SkipList using as the Element a class with `operator<` defined, thereby allowing the default comparison class implementation to make use of it?
- Can one create a SkipList using as the Element a class with no `operator<` defined? In this case, of course, the default comparison class must be overridden.
- Can we partially instantiate SkipList for an Element class with no `operator<<` defined for printing, so long as we do not call `printLevels`?
- Does the SkipList implementation use only the element inserted into the list is for comparisons, rather than the element from which the inserted element has been copied?
- Does the SkipList implementation create exactly one comparison class instance for a SkipList instance? This aspect enables the comparison class instance to be stateful, while the alternative of creating a new comparison class instance every time two elements must be compared does not.

Questions

Provide answers to these questions along with your solution when you turn in the assignment.

1. In `partial.cc`, why does the ALPHA SkipList insertion allow integer insertions?
2. Why do `timed` and `timedset` output different values for the same run if you don't use `rand_r` in the main function?
3. Use `time(1)` to measure execution time for insertion/deletion using your SkipList implementation and an STL set. The `timed` program uses your implementation, and `timedset` uses an STL set. Record values for each power of 2 in at least the range 2^{14} through 2^{20} and plot the results on a log-log plot (try gnuplot, for example).

   You should be able to see the N lg N behavior in your results. Estimate the constant factor for your skiplist as well as the STL set, report the estimated values (and mention how you calculated them), and add fit lines to your graph (you should have four lines and a legend on what you turn in).

   You may notice some discrepancy between the fit lines and your measured results, since your skiplist probably grows (and shrinks?) dynamically.

   Now, using the second command-line argument for calls to find, and 250,000 insertions, do the same thing for calls to find. Here the per-operation cost should be a constant, since your skip list is static at 250,000 elements. Note that you'll have to subtract out the cost of the fixed number of insertions/deletions for each type before calculating the cost of find operations. Show the same graph, again with four lines, report your estimate of the cost of find, and explain your methodology for calculating it.
4. The `distribution` program measures a simple random process. In a space of 128 (RAND_RANGE) integers, we repeatedly insert two values at random, then erase the minimum value. The program starts with some number of rounds to warm up the system, then counts the minimum values extracted in some additional number of rounds. When the experiment finishes, the program prints a histogram of probabilities for each possible minimum value in the skiplist.

   Run the program for a large number of rounds (perhaps ten million, although how much you can tolerate may depend on your SkipList's performance) in the default range (128 integers) and look at the results. Try to explain the shape of the curve. You should be able to predict the probability of erasing the value 0 exactly, but try to explain the downward trend after 0 and the phase shift as well.