Fill'er Up: Winners and Losers for Filling an Ordered Container

David Kieras & Steve Plaza, EECS Department, University of Michigan

A handy and common component is an ordered container, one that contains items maintained in some order such as alphabetical. In some applications, keeping the container in order as items are added to it is useful, such as in a spell-checking program: after a new word is added to the dictionary, the search can be done efficiently in the rest of the document. This article considers only this case, and the items are assumed to be unique.

Some STL containers, the associative containers, set<> and map<>, are inherently ordered - they store the contained items in a binary search tree (actually a red-black tree) where each new item is automatically inserted in the correct place in the container. As a result, searching the container is extremely efficient and iterating through the container always produces the items in order. In contrast, the sequential containers, list<> and vector<>, can store items in arbitrary order, so if an in-order container is desired, the programmer must write additional code to take advantage of the in-order contents for efficient search, and then add the item to the container in the proper position. The programmer has some choices in the specifics of how to write this code.

The purpose of this article is to compare the performance of different methods for filling an ordered container. Factors that affect how long it takes to fill a container include (1) the specific algorithm used to determine the correct location of the new item; (2) whether other items must be moved around to put the new item into the correct place; (3) how long it takes to compare individual items; and (4) if copying items is required, how long it takes to copy individual items. The analysis in terms of complexity normally considers only the first two factors. In practice, it is hard to know how these four factors will trade off against each other, but by comparing performance times it is possible to get some practical guidance. A set of comparisons were done by measuring the time required to fill containers of different types with different algorithms and with items that were either cheap or expensive in terms of the time to compare and copy. These times were collected for a wide range of number of items in the container.

The *Cheap* objects had a single integer member variable uniquely assigned when the object was constructed. Copying an object and comparing two of them involved simply this one integer variable. The *Expensive* objects contained a std::string member variable that at construction was given a value of the 26 alphabetical characters a-z with an additional five characters corresponding to the ASCII representation of a unique integer like that for the cheap objects. Thus copying these objects required copying a string, with its memory allocation cost, and comparing them required comparing a long string the first 26 characters of which were always identical. The classes for Cheap and Expensive objects defined the constructor and operator< as in-line functions, along with a get_key() function that returned the member variable value for use in map containers. The class declarations are shown in Box 1 below.

The execution time tests were done as follows in pseudocode:

1. Create a std::vector of *N* Cheap or Expensive objects.
2. Start the clock.
3. Do the following process *n_repeats* times:
    1. Use std::random_shuffle to put the vector in random order.
    2. Create an empty ordered container of the tested type.
    3. For each item in the randomly shuffled vector:
        add it to the ordered container using the tested method.
4. Stop the clock and output the execution time.

The container was always a stack object default-created within the scope of the for loop corresponding to Step 3. The filling algorithm always used the definition of operator< defined for the Cheap and Expensive objects. Note that the random_shuffle algorithm would be linear in time with number of items being randomized.

```
class Cheap {
public:
    Cheap(int i_ = -1) : i(i_) {}
    typedef int key_type;
    key_type get_key() const {return i;}
    bool operator< (const Cheap& rhs) const {return i < rhs.i;}
private:
    int i;
};

class Expensive {
public:
    Expensive(int i_ = -1)
        {
        ostringstream oss;
        oss << "abcdefghijklmnopqrstuvwxyz" << 10000 + i_;
        s = oss.str();
        }
    typedef string key_type;
    key_type get_key() const {return s;}
    bool operator< (const Expensive& rhs) const {return s < rhs.s;}
private:
    string s;
};
```

Box 1. Class declarations used in the run-time comparisons.

The container and algorithm combinations (called *methods* hereafter) shown in the table below were tested for both Cheap and Expensive objects, using the code fragments shown for inserting each new item into the container. The set container can only be filled with its insert member function. The map container can be filled either by creating a pair and inserting it, or by using the subscript operator. The get_key() function simply returns the value stored in the object. A list container can only be searched linearly, but the search can terminate as soon as the point of insertion is located, and then the new item inserted at that point. Alternatively, the new item can be pushed onto the end of the list, and then the entire container sorted using its member function. Since a vector container has random-access iterators, a fast binary search (with std::lower_bound) can be performed to locate the insertion point, and then the new item inserted at that point. In an unfortunately common student practice, the new item can also be pushed onto the end of the vector, and the std::sort algorithm applied.

Method Name	Container	Item insertion code
Set	`set`	`cont.insert(item);`
MapMakePair	`map`	`cont.insert(make_pair(item.get_key(), item));`
MapSubscript	`map`	`cont[item.get_key()] = item;`
ListLinInsert	`list`	`it = find_if(cont.begin(), cont.end(), bind1st(less<T>(), item));` `cont.insert(it, item);`
ListPushSort	`list`	`cont.push_back(item);` `cont.sort();`
VecBinIns	`vector`	`it = lower_bound(cont.begin(), cont.end(), item);` `cont.insert(it, item);`
VecPushSort	`vector`	`cont.push_back(item);` `sort(cont.begin(), cont.end());`

In the results shown below, the values of *N*, the number of items to be inserted into the container, were {10, 50, 100, 500, 1000, 5000, 10000, 50000, 100000}, and *n_repeats* was 100. The clock times were recorded using the process times provided by `getrusage`. Some of the methods were so slow that only the smaller values of *N* were used, so not all methods have all data points, as will be shown in the results. The compiler/library used was gcc 4.7, with std=c++11 and -Os optimization level, and run on CAEN login server workstation at the time of this writing.

Results

The run times have a very large range, so the results will be shown separately for cheap objects and expensive objects, and with multiple graphs for each - one the covers the entire range produced by the slowest methods, and additional ones that zoom into the much smaller range needed by the fastest methods.

Cheap Objects

Figure 1 below shows the time required by the methods for Cheap objects, with the run time for 100 repetitions on the y-axis and *N* on the x-axis. Each method is plotted separately. The y-axis covers the full range of run times, shown on a 700-sec scale that will be used later for the Expensive objects. It is obvious that two of the methods are very much worse than the others; these are the two pushback+sort methods VecPushSort and ListPushSort, with the List method being the worst. For 10,000 Cheap items, these methods take more than 1000 times longer than the fastest method, which is hidden in all the other methods that take very little time in comparison.

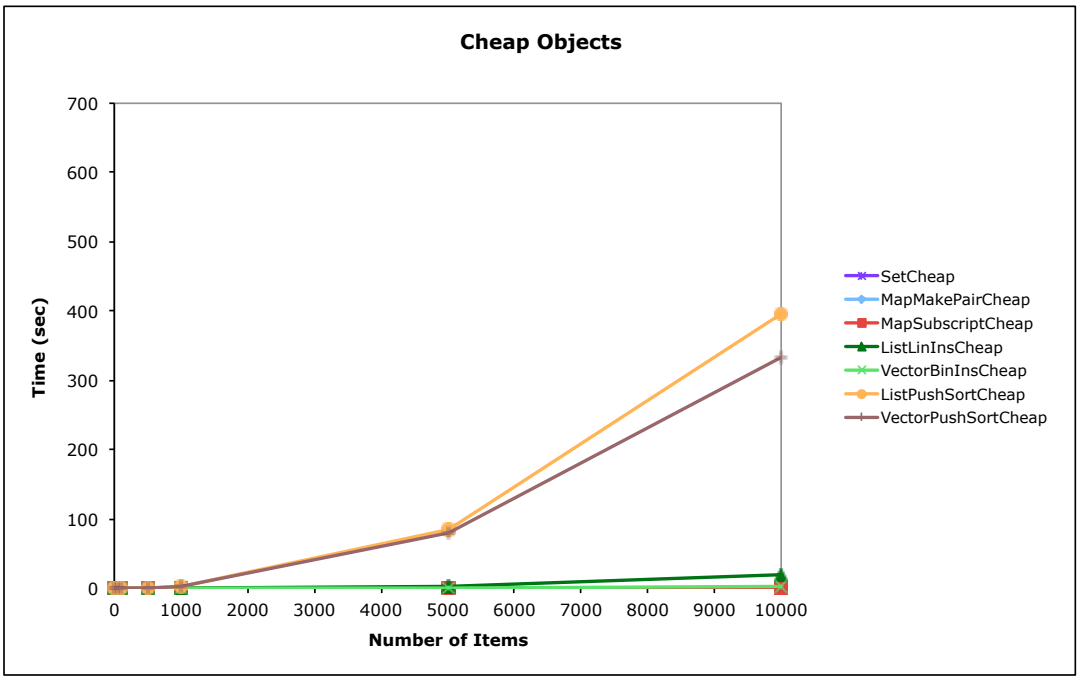

Figure 1. Full-range results for Cheap Objects.

Figure 2 (below) shows the same data but zoomed in on the y-axis to cover a range of only 0-10 seconds, to start to show the differences between the fast methods that all plot on top of each other in the first graph. The skyrocketing times for the PushSort methods are obvious here. In stark contrast, the Set method and the two Map methods take almost no time. We'll look at them more closely in the next graph. The ListLinIns method increases sharply, as we would expect from the linear search being applied to an ever-lengthening list. The VecBinIns method fares considerably better due to the fast binary search to locate the insertion point, but the fact that a lot of data movement is involved in the vector container will hurt when we look at expensive objects.

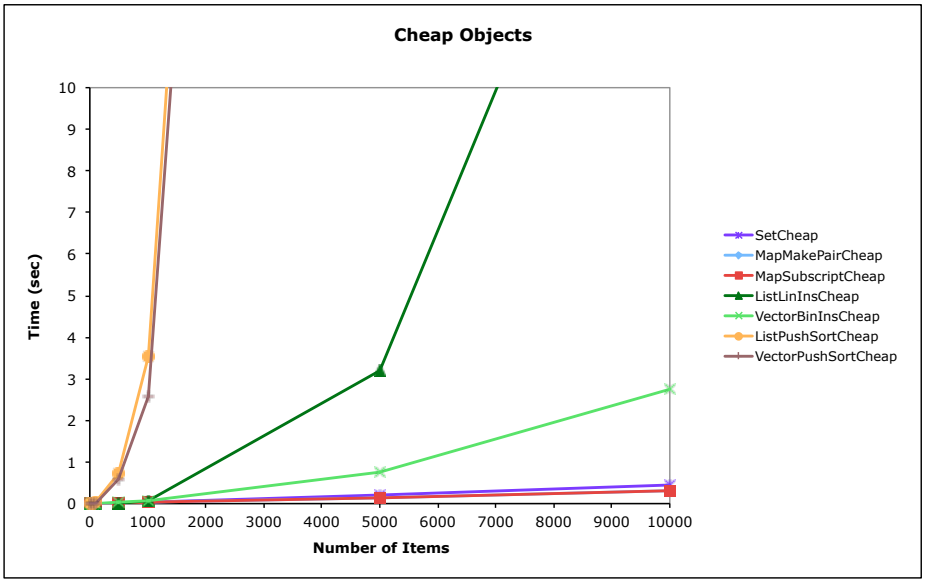

Figure 2. Zoomed-in results showing the first 10 sec for Cheap objects.

Figure 3 zooms in even closer to show the Set and Map methods and includes some additional runs of 50,000 and 100,000 items (whose run times with the other methods were too long to be sensible to collect). These methods look almost linear with the number of items; the very rapid logarithmic search times is combined with the time to copy each item into the container, which contributes a strong linear component. The positive acceleration in the run time is barely visible in this graph even with 100,000 items being stored. In these results, the Set method and MapMakePair are essentially identical, while MapSubscript has a slightly larger slope because using the map subscript operator to fill the container requires constructing and then copying over an empty object in the container for each new item. Even so, the increase in run time is trivial compared to the methods using sequence containers.

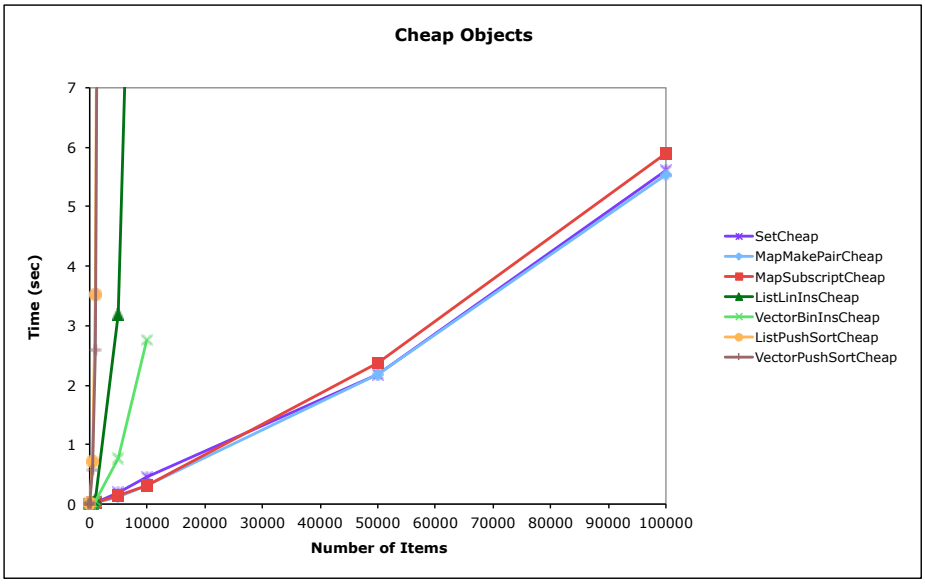

Figure 3. Zoomed-in results showing the first 7 sec and N = 100,000 for Cheap objects.

Does the speed advantage of map and set remain even for a small number of objects? The more sophisticated containers involve some overhead that might be a relatively large part of the run time if only a few objects are involved. Figure 4 shows really-zoomed-in results, covering only up to 5000 objects and 3 sec of run time. Clearly, at N = 5000, map and set are winning, but they don't look better at smaller sizes.

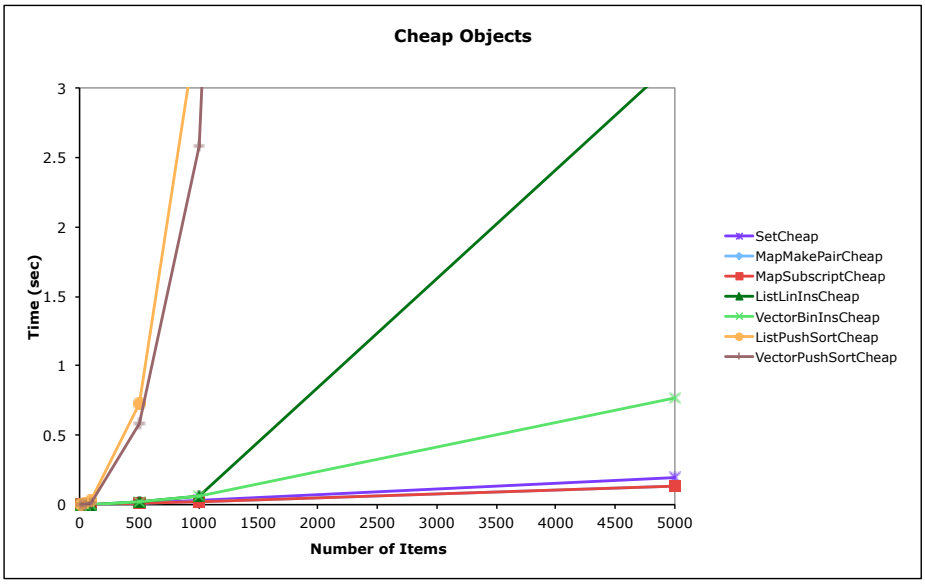

Figure 4. Zoomed-in results showing N <= 5000 for Cheap objects.

Let's look closer. Figure 5 is hyper-zoomed-in, showing *N* up to 500 and time up to 0.02 sec. Note that at these small values, the timing values become noisy; more extensive run tests would be needed to distinguish any differences. At *N* = 100 and below, the differences between the fast methods become very slight; even the map container does well. So if you have only a small number of objects that are cheap to compare and copy, using a vector or list with a find-and-insert method can make sense. But note that the PushSort methods are still losers even in this range!

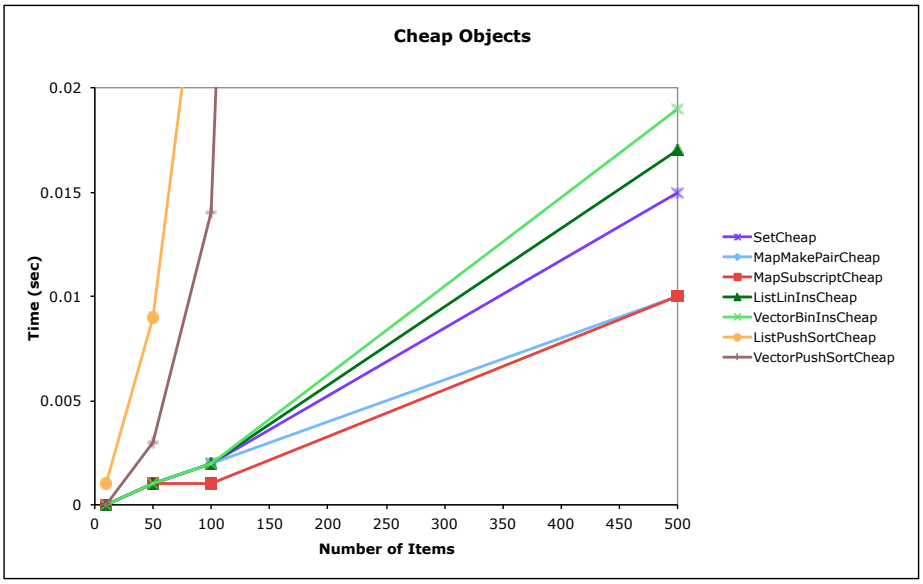

Figure 5. Zoomed-in results showing N <= 500 for Cheap objects.

Expensive Objects

Expensive objects take much longer to compare and copy than the Cheap objects. Figure 6 shows the full range for the methods on the same 700-sec scale as Figure 1. Again, the two PushSort methods are very much slower than the others, increasing so rapidly that their run times were determined only up to $N = 5000$. VecPushShort is catastrophically slow due to the the greater cost of copying the objects during the data movement. Though still on a fast upward trend, ListPushSort method is better because the list requires only pointer-jiggling to insert rather than moving half (on the average) of the items already in the container. Again, the two search-and-insert methods ListLinIns and VectorBinIns are considerably better than the push-sort methods, but rise much more steeply than the Set and Map methods which stay flat at near-zero by comparison.

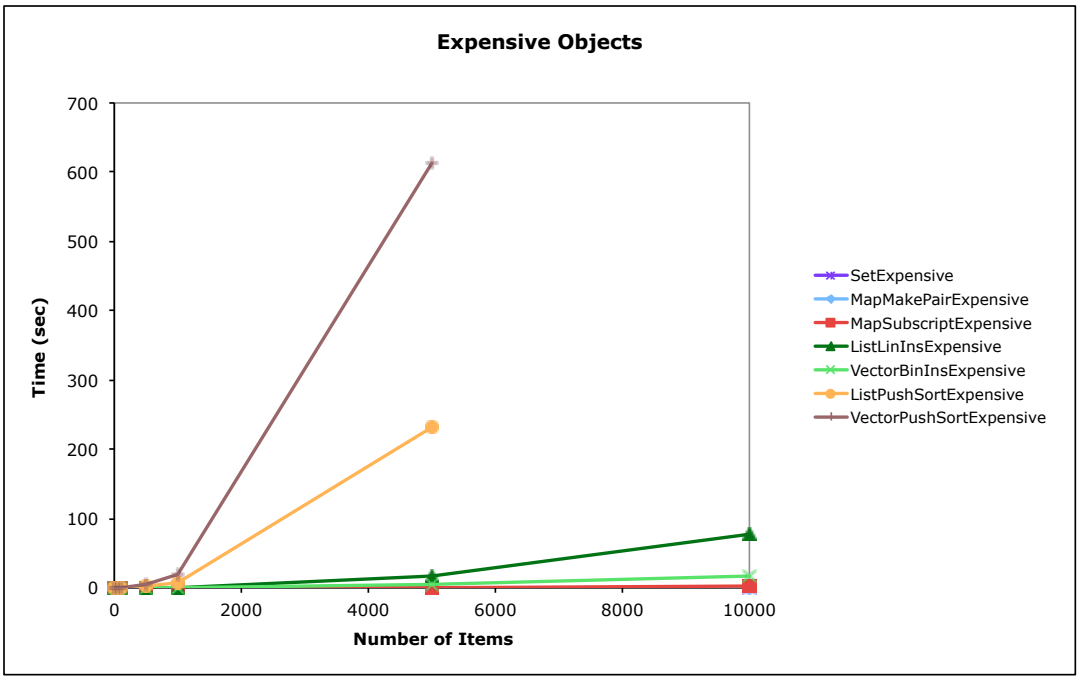

Figure 6. Full-range results for Expensive objects.

Let's zoom in on the faster methods again, to cover only a 2-second range. These results are shown in Figure 7. Again, the PushSort methods are blasting off into the wild blue yonder compared to the other methods - what losers! The two methods using a sequence container, ListLinIns and VectorBinIns are good only by comparison to the real losers, but they are also radically increasing. The winners all look almost linear: Set and MapMakePair are clearly the fastest methods, meaning that at least for objects like these, there is no reason to insist on using a set if a map would be more suitable. The greater overhead of using the subscript operator in MapSubscript suggests that this method should only be used for convenience in one-time-only sorts of operations, such as initializing a container for a small number of items.

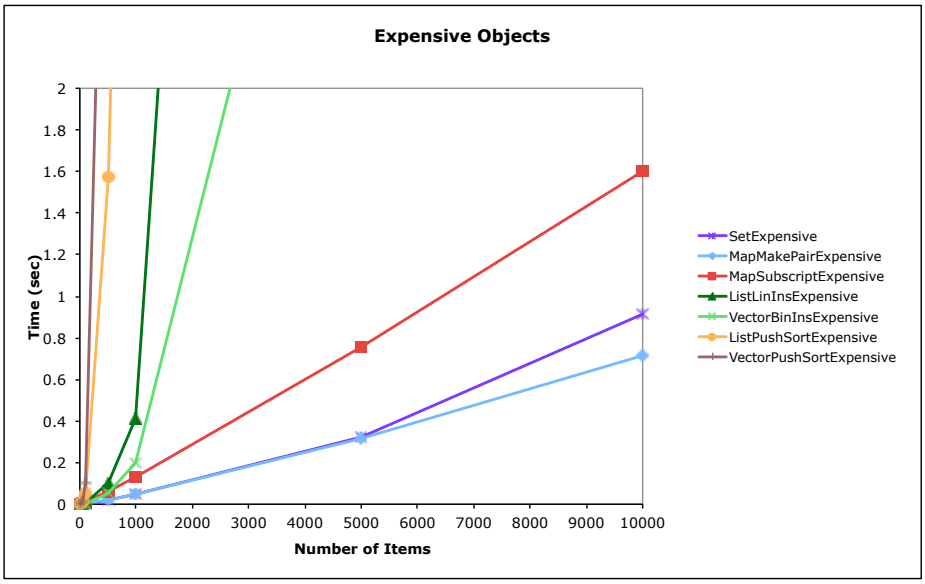

Figure 7. Zoomed-in results showing the first 75 sec for Expensive objects.

What about for very small values of *N*? Figure 8 shows hyper-zoomed in results for N up to 100. Again we are down in the weeds of noise with the fastest containers. Clearly there is not much difference for very small *N*, but even so, the pushback+sort methods start loosing very quickly. The interesting thing is that ListLinIns and VectorBinIns are competitive with the fastest methods at this small *N*, so there is some evidence that the greater overhead of the map and set containers might be an issue at small *N*.

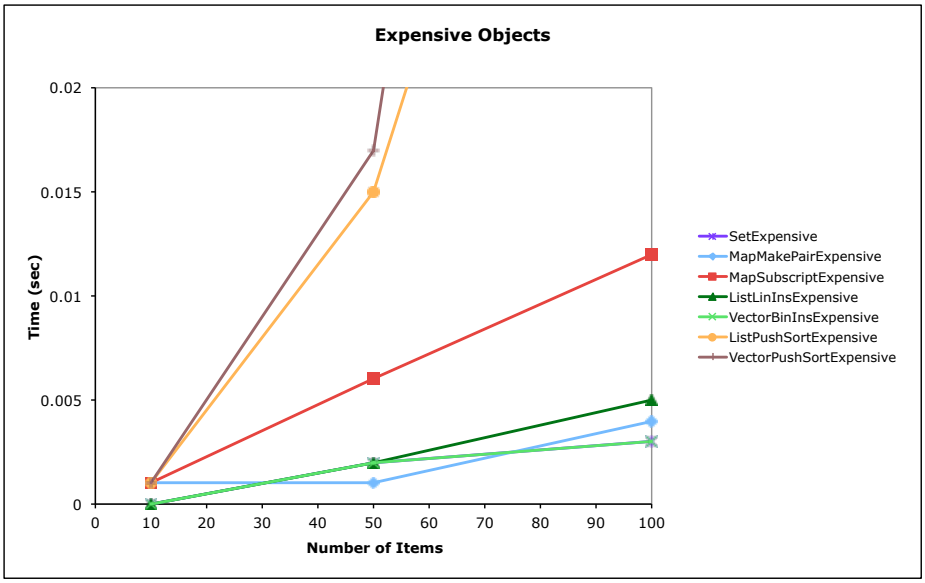

Figure 8. Zoomed-in results showing N <= 100 for Expensive objects.

What does Big-O Say?

Let's take a quick look at the complexity of the container-filling operation. First, under the constraints we are investigating here, where the items are to be added one at a time to a container that must be kept in sorted order, the *O* has to be at least *O(N)* because each item has to be added to the container, and even if each search and insertion could be done in constant time, the total time would be linear with the number of items to be added. To start with the best situation, if the search for where to insert each item can be done with *LogN* comparisons (as with the Set and Map containers), then to insert *N* items would take *N* such searches in the limit, giving *O(NLogN)*. Next best would be that the search would be linear, taking *N* comparisons for each of the *N* items; this would give $O(N^2)$ which clearly increases much faster with *N*. Finally, suppose each insertion requires a complete sort of the *N* items already in the container. std::sort applied to a std::vector container and std::list.sort() are specified as requiring *O(NLogN)* in the number of comparisons. However, the sort has to be repeated *N* times, resulting in $O(N^2LogN)$.

Let's compare these theoretical formulas graphically for the same range of *N* as used in the run-time testing. Figure 9 shows *N* ranging from 0 to 10,000, and the values of *NLogN*, N^2, and N^2LogN for these values of *N*. The results are similar to the run times. N^2LogN takes off like a rocket, just like the push_back+sort methods; N^2 is not as bad, but still increases very quickly, like the ListLinIns and VectorBinIns methods.

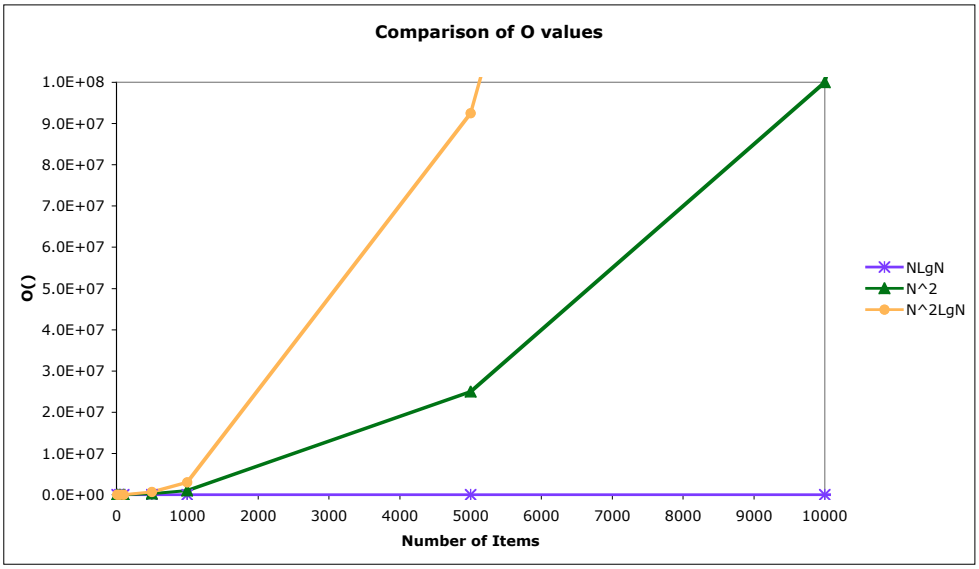

Figure 9. Wide-range results for O comparison.

Finally, *NLogN* by comparison is very small throughout the range. Figure 10 below zooms in on the vertical axis, and just like we saw for the Set and Map containers, *NLogN* increases slowly and almost linearly by comparison with the other methods which shoot up off the top of the graph.

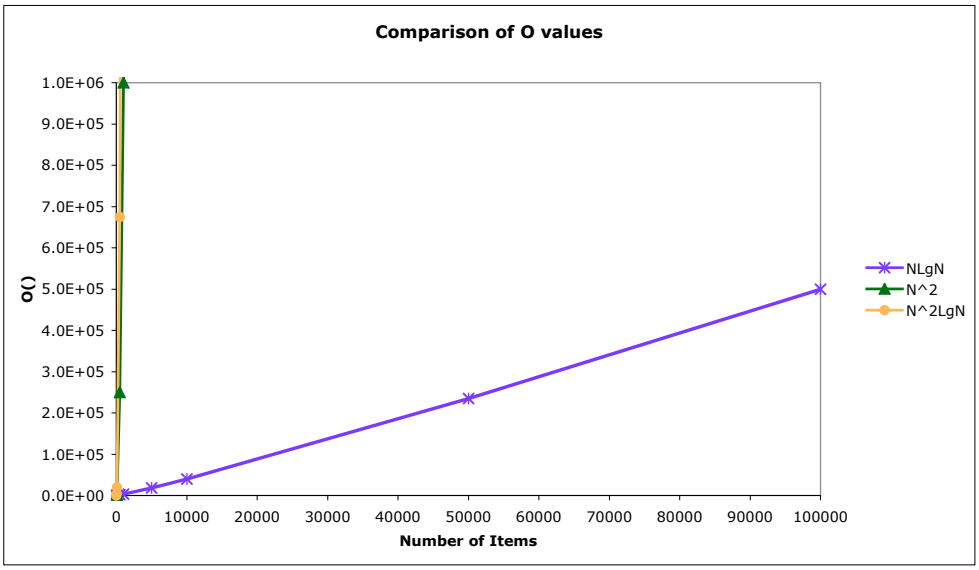

Figure 10. Zoomed-in results for O comparison.

Thus the Big-O analysis corresponds well to the observed run times; what is left out are some of the factors that would modify the slopes of these curves, such as the relative time to compare objects, copy objects, or insert an object into the container. These factors are important, especially at very small sizes, but generally they are really quite minor compared to the huge differences produced by differences in the abstract complexities of the different methods. While *O* doesn't tell the whole story, programmers who ignore what it has to say should expect nasty surprises from slow-running code.

Concluding Recommendations

Keeping in mind that your mileage might vary depending on your specific application, these results can be summarized with the following recommendations.

1. If a very small number of items are involved, filling a vector with binary search and insert or a list with linear search may be reasonably fast. However, larger sizes quickly make the set and map methods faster.

2. If a map container is convenient on other grounds, there is no reason not to use if it is filled with make_pair; reserve the subscript operator for non-speed critical applications where its greater legibility is valuable, such as initializing a map with a small number of constants.

3. If a sequence container is needed, use a vector container and fill it only with binary search and insert. Filling a list container with linear search and insert is generally slower in these results, even though inserting into a list involves less data copying than inserting into a vector. Big differences in the cost of comparing versus copying the objects might change this relationship in specific cases.

4. Filling a sequence container with push_back and sort is a ridiculously slow method even for cheap objects and relatively small sizes. Unless you have only a handful of items, it is going to run very slowly — *just don't do it!*