Why std::binary_search of std::list works, sorta ...

David Kieras, University of Michigan

Prepared for EECS 381, April 2006

You can indeed apply the Standard Library binary_search and lower_bound algorithms to any sorted sequence container, including std::list, and it will produce a correct result. However, if you look at any normal code for binary search (e.g. as in Kernighan and Ritchie), it is written to use array subscripting, which runs in constant time, independent of the size of the array. A linked list has the property that the only way you can find a node is to start at one end of the list and follow the links from one node to the next, checking them one at a time. Unlike with array subscripting, there is no way to compute the location of a list node directly from knowing its numerical position in the list. So how is it that you can do a binary_search of a std::list?

How binary_search is implemented. Below is a somewhat simplified copy of the Metrowerks Standard Library version of lower_bound; the binary_search algorithm just calls lower_bound and checks the result (other implementations might differ, but only in details).

```
template <class ForwardIterator, class T>
ForwardIterator
lower_bound(ForwardIterator first, ForwardIterator last, const T& value)
{
   typedef typename iterator_traits<ForwardIterator>::difference_type difference_type;
   difference_type len = distance(first, last);
   while (len > 0)
   {
      ForwardIterator i = first;
      difference_type len2 = len / 2;
      advance(i, len2);
      if (*i < value)
      {
         first = ++i;
         len -= len2 + 1;
      }
      else
         len = len2;
   }
   return first;
}
```

First, see how this algorithm is written in terms of iterators, so that it can apply to any sequence container that supports the standard iterator interface. The two *input iterator*s are *first* and *last*, marking the beginning and end of the range to be searched. In the type system, input iterators can be iterators pointing into any type of container. The basic binary search algorithm involves calculating the midpoint of a range of values, and then checking the value at that midpoint. The code calls std::distance, which returns the numerical distance between the first and last iterators. This distance is divided by two, and then std::advance is called to move the first iterator forward by that amount to get to the midpoint of the range. The distance and advance functions are also function templates that are defined so that they work with iterators into any type of container. Template magic is used to specialize them for different iterator types. For *random-access iterator*s (which behave like pointers or subscripts), which std::vector and std::deque supply, the definition that is used is:

```
template <class RandomAccessIterator>
inline
typename iterator_traits<RandomAccessIterator>::difference_type
__distance(RandomAccessIterator first, RandomAccessIterator last, random_access_itera-
tor_tag)
{
    return last - first;
}
```

The subtraction operator is defined for these iterators because the internal pointers can simply be subtracted to get the distance directed via pointer arithmetic. However, for the more general input iterators, which can only move forward or back by one step at a time, the definition is:

```
template <class InputIterator>
inline
typename iterator_traits<InputIterator>::difference_type
__distance(InputIterator first, InputIterator last, input_iterator_tag)
{
    typename iterator_traits<InputIterator>::difference_type result = 0;
    for (; first != last; ++first)
        ++result;
    return result;
}
```

In other words, compute the distance between two general iterators by incrementing the first until it equals the last, and count how many increments are required.

The advance function has a similar pair of specializations. Advancing a random-access iterator simply adds the number of steps to the iterator, corresponding directly to pointer arithmetic, but advancing an input iterator requires incrementing the iterator the supplied number of times.

When these functions are applied to a built-in array or a std::vector container, the distance and advance functions will compile down to simple subtraction and addition of the subscript values or pointers, taking almost no time. But if applied to a list, whose iterators support only moving forward or back by one step at a time, then the binary search will require using the distance function to repeatedly count the nodes between the ends of the narrowing range and then advance with increment and count again to position at the midpoint. Surely all this link-following will add up to a substantial amount of time, won't it?

The Standard in fact states that lower_bound and binary_search will run in *O(log n)* time when applied to a container with random access iterators. When applied to a container that lacks random-access iterators, like std::list, the Standard states that the search will be logarithmic with the number of comparisons, but linear with the number of nodes visited. So whether it runs faster than a linear search depends on how the much time it takes to do the comparisons compared to the counting the nodes over and over again.

Big-O doesn't tell the whole story. Sometimes you need benchmarks or profiling to see how things work out in practice. I defined two classes of objects which contain an ID value used in operator< and operator==, and with constructors that give each object a unique value. One class, Cheap, uses a single integer for the comparisons. The other, Expensive, uses a string containing 31 characters, the first 28 of which are identical. The comparison is done with std::strcmp, which will have to test the first 28 characters before finding the different ones. My benchmark code filled containers with 10, 50, 100, 500, or 1000 objects, and then ran a binary search looking for each one of the objects, so each possible position in the container was searched for. This search sequence was repeated more for shorter containers so that the final results show the total time for 50,000 searches distributed evenly over all the positions in the container, giving reasonably stable average run times. I compared run times for the eight combinations of std::vector vs. std::list containers, linear search with std::find vs. binary_search, and Cheap vs. Expensive objects.

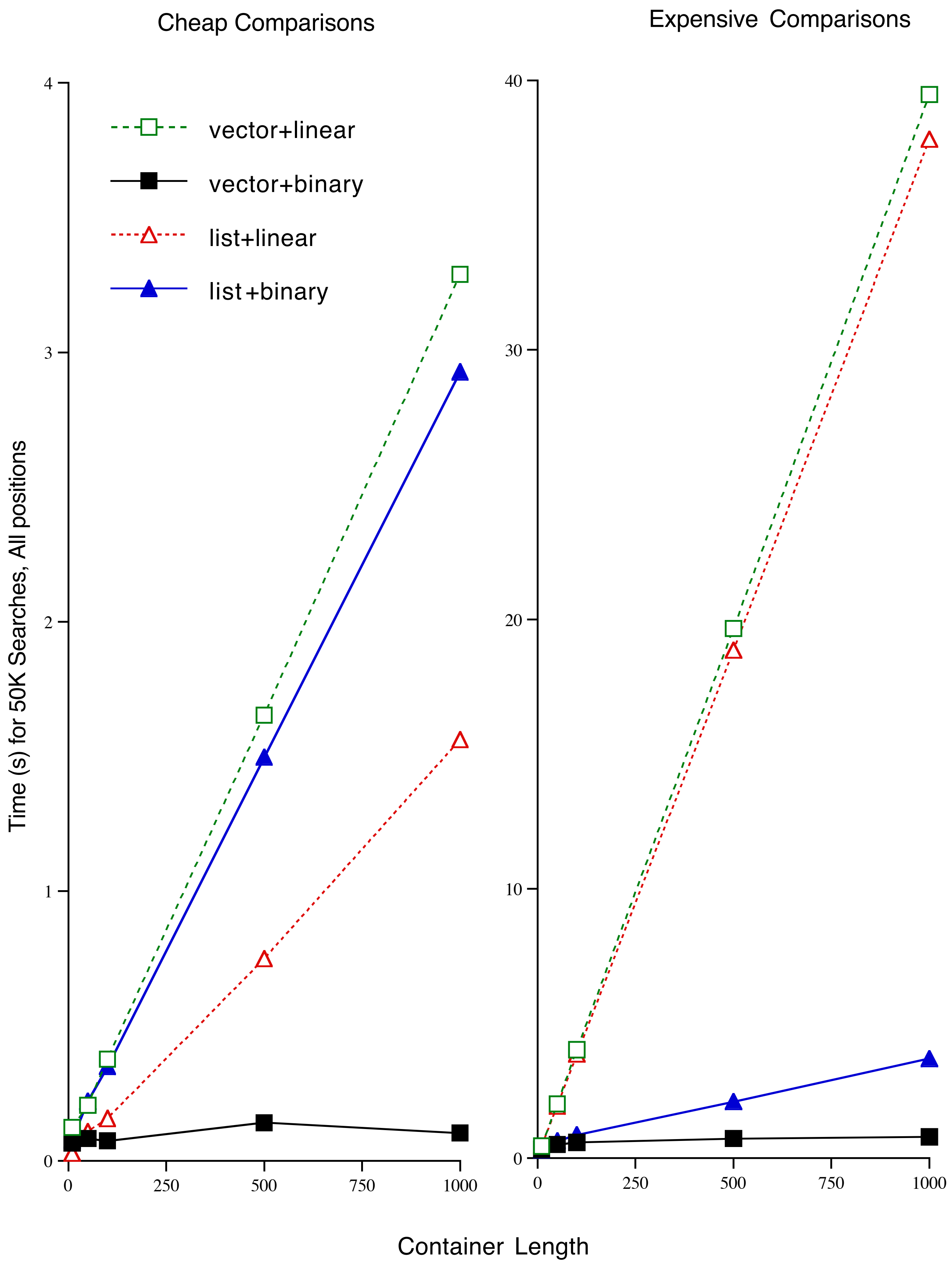
The graphs on the next page show the results. First, the Expensive object searches required about ten times longer than the corresponding Cheap object searches. Searching vector linearly is the big loser, while using binary_search on vector is so fast that on the plotting scale it looks almost flat - this really is the best combination. A surprise is that for Cheap objects, the linear search of a list ran substantially faster than the linear search of a vector. Apparently, the constant-time arithmetic for doing the subscripting is slower than merely following a pointer to the next node. This effect is almost swamped by the time required for the comparisons in the Expensive objects; the two linear searches have similar speeds.

The key result for this discussion is how for the std::list container, the effects of doing linear vs. binary searches depends on the cost of the comparisons. Compare the two curves plotted with triangles in the graphs.

For Cheap objects, the linear list search is faster than binary list search by about twice - using binary_search would be a serious mistake in this case. In fact, it is almost as slow as the linear vector search. The binary list search requires counting those nodes over and over again, and this can really add up.

For Expensive objects, the binary list search is quite a bit faster than the linear list search which is almost as slow as the linear vector search. The way in which binary search minimizes the number of comparisons really pays off. However the binary list search looks pretty much linear, rather than logarithmic, and is still substantially slower than the binary vector search. Counting nodes back and forth in the list still hurts substantially, even if we save a lot by doing only a logarithmic number of comparisons.

The bottom line: Of course, your mileage will vary depending on the cost of comparing your objects and the lengths of your lists and the distributions of your searches. However, together with purely theoretical considerations, these results show that you need really good reasons to search a list with binary search, and one of those reasons needs to be that the objects in the list are quite expensive to compare. It really is true that if you want to search an ordered sequence container quickly, you can't beat binary search of a vector.


Cheap Comparisons
Expensive Comparisons
vector+linear
vector+binary
list+linear
list+binary
Time (s) for 50K Searches, All positions
0
1
2
3
4
0
10
20
30
40
0
250
500
750
1000
Container Length