Mitchell Gouzenko
Library Design C++
Homework 2

Introduction
The purpose of this homework was to test the speed of insertion and deletion in vectors and linked lists in the experiment specified by professor Stroustrup. Here, I will highlight my findings, surprises, and subsequent experiments that I performed to justify the results.

Approach
I inserted random elements (in proper order) into both vectors and linked lists, and subsequently emptied the containers by deleting elements at random indices.

To do so, I wrote a function called **generate_fillers** that creates a vector of random integers from a uniform distribution. These fillers are subsequently fed into vectors and linked lists, inserted in the correct positions. The function **generate_removal_indices** creates a vector of random positions. The positions are used to remove elements from the vectors and linked lists.

I timed this whole process of insertion and deletion for both vectors and lists.

Inserting elements
Since the assignment specifications called for the same algorithm for both linked lists and vectors, I decided to use a linear method of insertion. This is because it does not make sense to perform a binary search on containers unless they support random access – which the list is incapable of.

Hypothesis
When I started this assignment, I simply assumed that the vector would outperform the linked list for a low number of elements. Then, at some point, the linked list would start outperforming the vector. My logic was as follows: vectors incur an O(n) cost for insertions, since they store underlying elements in arrays. On average, when an element is inserted into its appropriate position, half of the other elements need to be shifted over by one spot to make space for the new element. For low n, there is not much overhead to do this. But, for high n, many elements need to be moved, so insertion should be slow.

Linked lists, on the other hand, perform insertions in constant time – assuming that an iterator that points to the desired insertion location is readily available. Thus, for large n, the vector's insertion time should grow linearly, while the linked list's remains the same.

In total, the time complexity for vectors should be $O(n^3)$, dominated by element insertion. For linked lists, the time complexity should be $O(n^2)$:

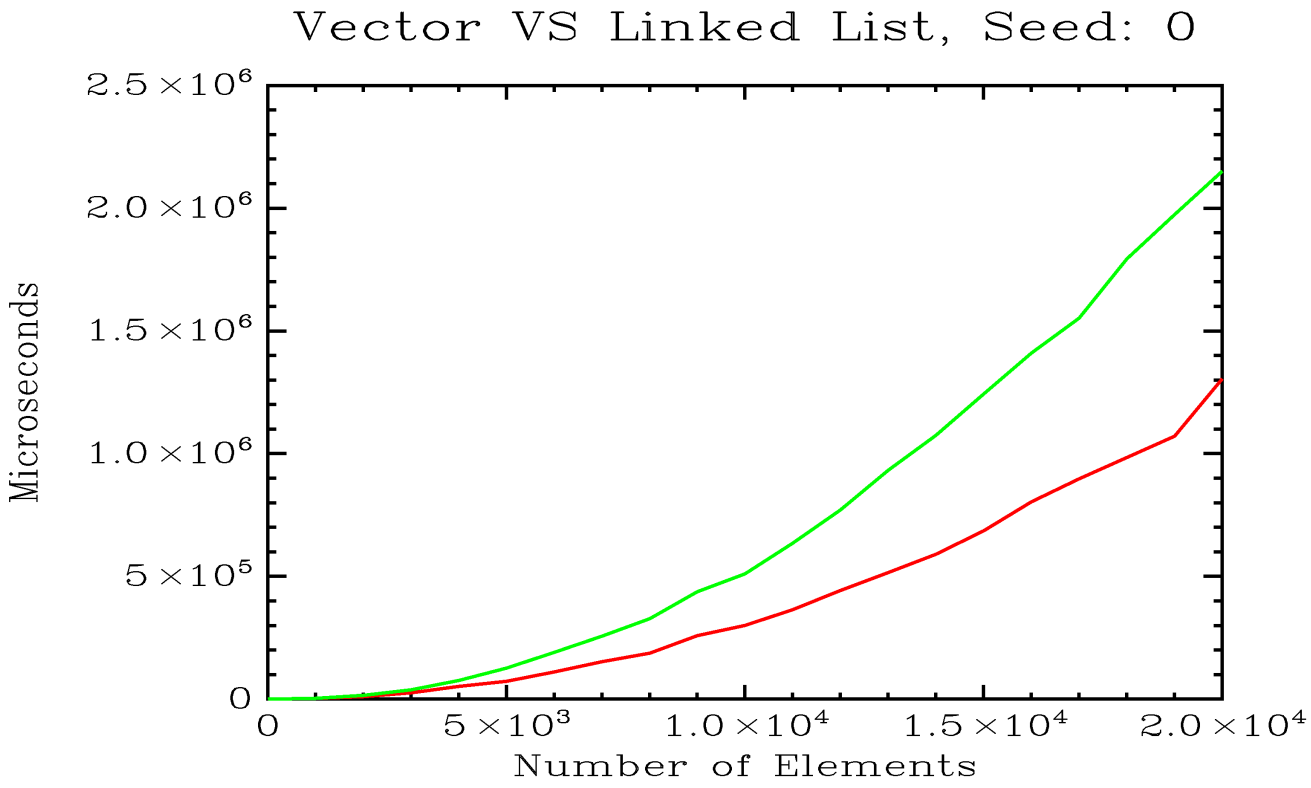
$$\begin{aligned} \boldsymbol{Vector}: &\quad n\ elements * O(n)\ traversal * O(n)\ insertion = O(n^3) \\ \boldsymbol{Linked\ List}: &\quad n\ elements * O(n)\ traversal * O(1)\ insertion = O(n^2) \end{aligned}$$

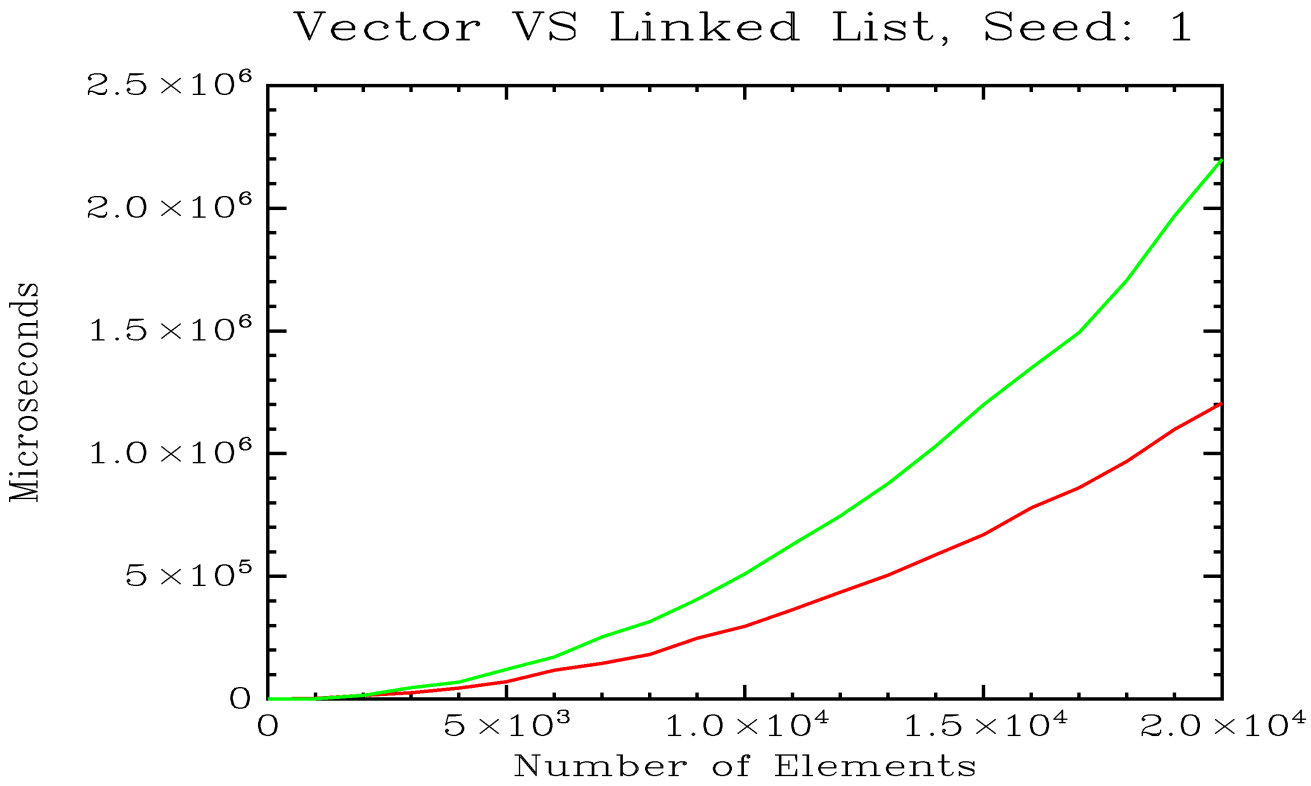
Because of this difference in complexities, the linked list should eventually outperform the vector.

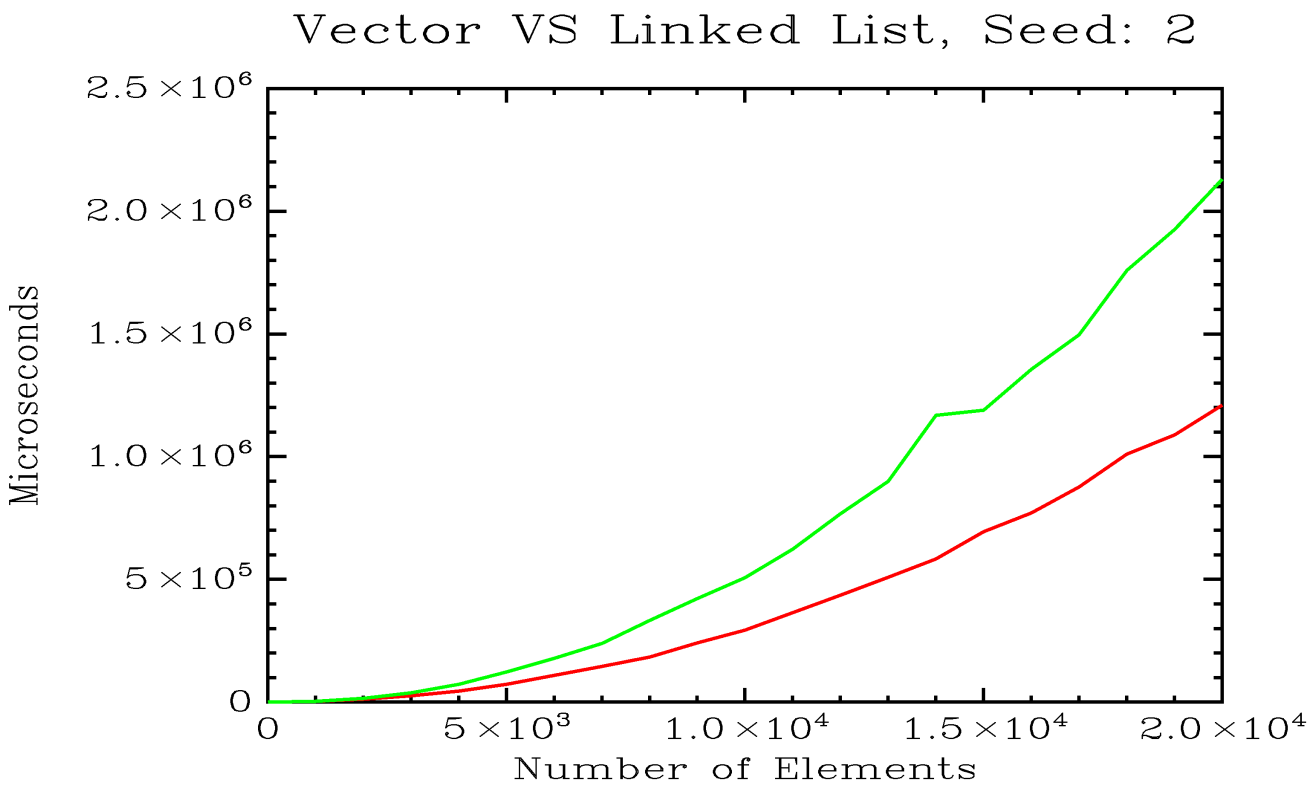
Results
The results of the experiment are shown below for various seeds. The green line represents the linked list, and the red line represents the vector. The vector **consistently outperformed the linked list**. This is contrary to my hypothesis.

Note: Vector is RED
List is GREEN

All graphs have a step of 1,000







Analysis
The vector's superiority over the linked list must be somehow accounted for. After discussing my surprising results with several classmates, I formed the hypothesis that vector operations were being optimized through usage of the cache. That is, when a portion of the vector is accessed in memory, the memory locations surrounding it are pre-fetched into cache. Then, since we're accessing data sequentially, most of the shifting that occurs during insertion happens within the cache.

For linked lists, on the other hand, the nodes are scattered across memory instead of being stored contiguously. The list's runtime is **dominated by traversal**. That is, to get from one node to the next, the list has to access that next node's address from memory. Because the nodes are not contiguous in memory, the next node is probably not pre-fetched into cache. Thus, traversing a linked list **maximizes cache misses.**

To explore this hypothesis, I ran the speed test through Valgrind for lists and vectors separately. In this test, I populated and subsequently emptied a 10,000-element vector, and then – in a different experiment – a 10,000 element list. I used Valgrind's "callgrind" tool, which analyzes cache usage and reports cache misses. The results are below.

Linked List Cache Report

```
==15959==
==15959== Events    : Ir Dr Dw I1mr D1mr D1mw ILmr DLmr DLmw
==15959== Collected : 1853321970 709433891 407532996 2233 41280477 97399 1885 4449 5832
==15959==
==15959== I   refs:      1,853,321,970
==15959== I1  misses:            2,233
==15959== LLi misses:            1,885
==15959== I1  miss rate:           0.0%
==15959== LLi miss rate:           0.0%
==15959==
==15959== D   refs:      1,116,966,887  (709,433,891 rd + 407,532,996 wr)
==15959== D1  misses:       41,377,876  ( 41,280,477 rd +      97,399 wr)
==15959== LLd misses:           10,281  (      4,449 rd +       5,832 wr)
==15959== D1  miss rate:           3.7% (        5.8%   +         0.0%  )
==15959== LLd miss rate:           0.0% (        0.0%   +         0.0%  )
==15959==
==15959== LL refs:          41,380,109  ( 41,282,710 rd +      97,399 wr)
==15959== LL misses:            12,166  (      6,334 rd +       5,832 wr)
==15959== LL miss rate:            0.0% (        0.0%   +         0.0%  )
```

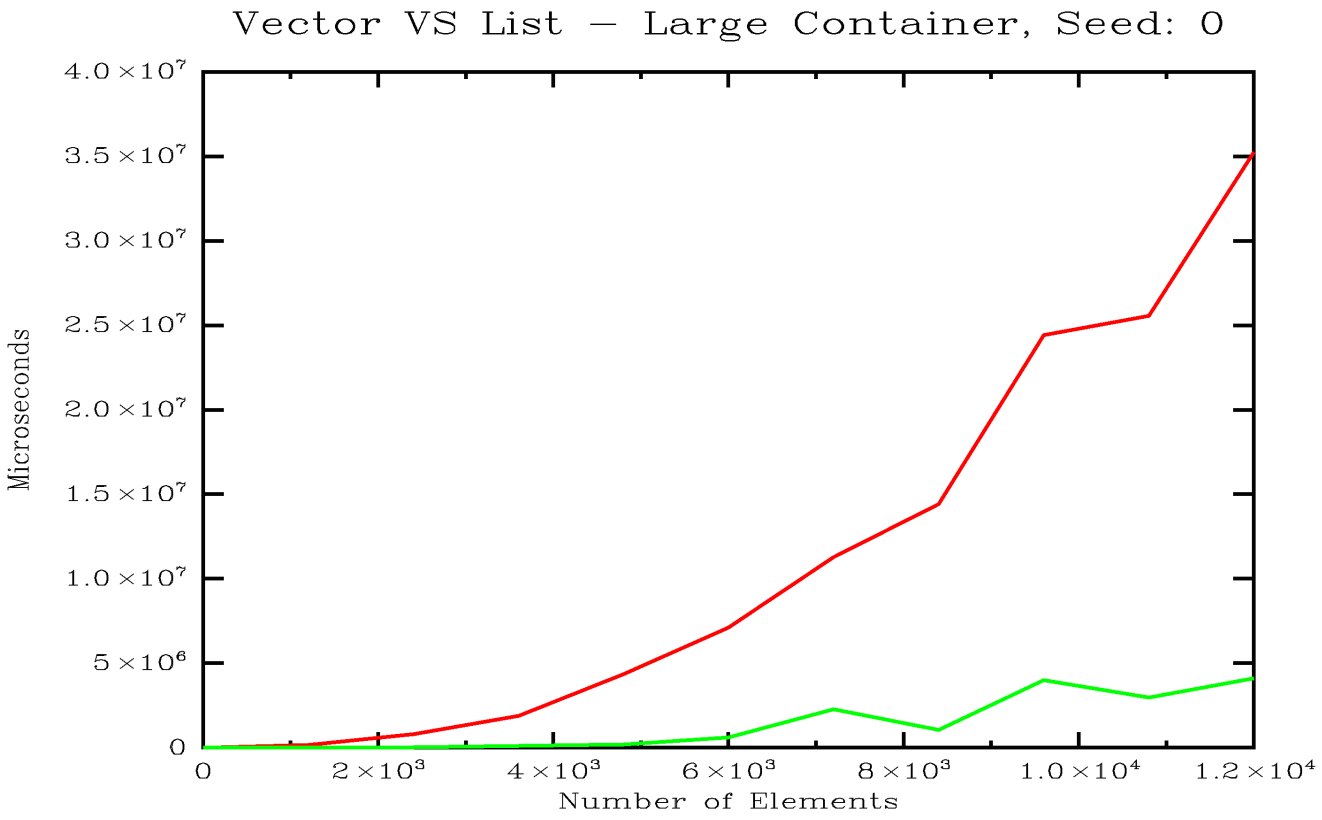
Vector Cache Report

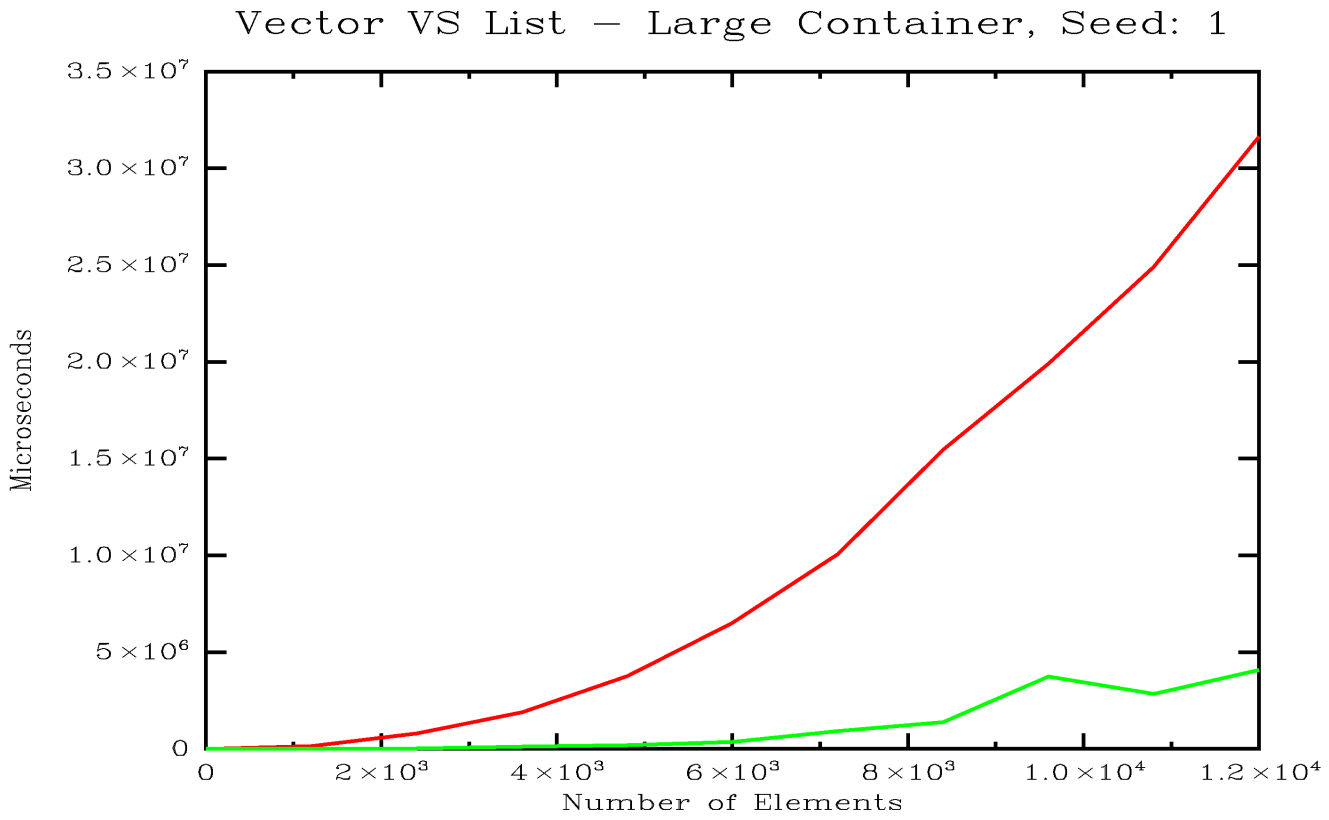
```
==16099==
==16099== Events    : Ir Dr Dw I1mr D1mr D1mw ILmr DLmr DLmw
==16099== Collected : 2366018033 851558150 599006255 2321 661862 12545 1940 4450 5128
==16099==
==16099== I   refs:      2,366,018,033
==16099== I1  misses:            2,321
==16099== LLi misses:            1,940
==16099== I1  miss rate:           0.0%
==16099== LLi miss rate:           0.0%
==16099==
==16099== D   refs:      1,450,564,405  (851,558,150 rd + 599,006,255 wr)
==16099== D1  misses:          674,407  (    661,862 rd +      12,545 wr)
==16099== LLd misses:            9,578  (      4,450 rd +       5,128 wr)
==16099== D1  miss rate:           0.0% (        0.0%   +         0.0%  )
==16099== LLd miss rate:           0.0% (        0.0%   +         0.0%  )
==16099==
==16099== LL refs:             676,728  (    664,183 rd +      12,545 wr)
==16099== LL misses:            11,518  (      6,390 rd +       5,128 wr)
==16099== LL miss rate:            0.0% (        0.0%   +         0.0%  )
```

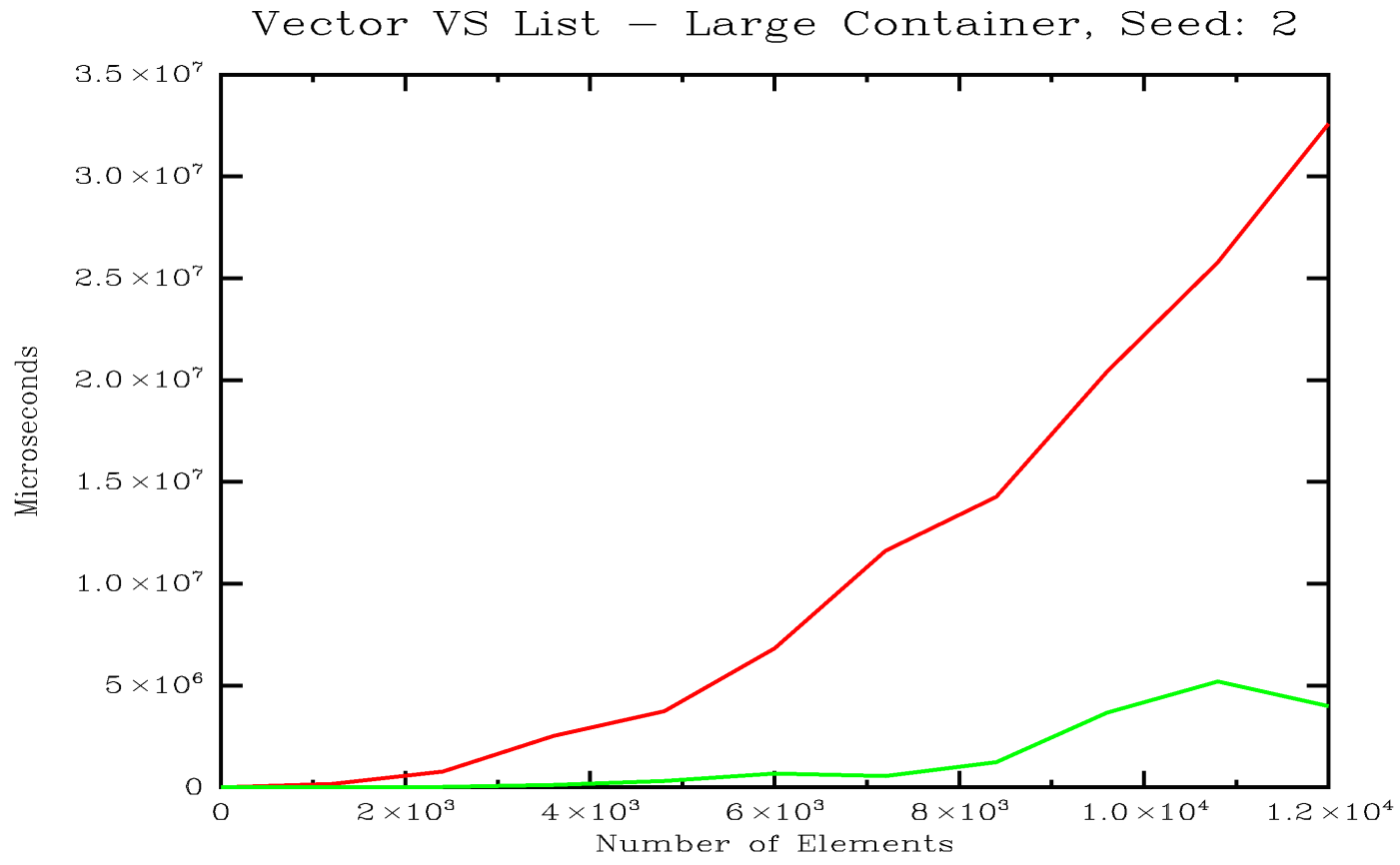
An important discrepancy to notice between the two reports is the D1 miss count. The D1 cache is the L1 cache's Data Cache – separate from the Instruction Cache (I1). The linked list yielded 41,377,876 D1 cache misses, while the vector yielded only 674,407 cache misses. That is, the linked list had approximately **61 times the cache misses** of the vector. Furthermore, that figure is expected to grow with the number of elements in the experiment.

Further Experimentation
These results beg the question: in what circumstances should linked lists be used over vectors? To explore this question, I tried to cripple the performance of the vector. I performed the same experiment, but instead of inserting and deleting integers, I inserted and deleted elements of my custom class, **large_container.** Large container simply contains an integer as a private member, as well as a 1000-byte char array. I hypothesized that the effect of this handicap would be twofold: firstly, the vector will need to shift many more bytes over to insert in the middle of the sequence. Thus, we expect memory reads and writes to increase. Furthermore, for sufficiently large n, the vector will not fit in cache. So, we can expect cache misses to increase. Both of these factors should make the vector perform worse than the linked list.

Results
As expected, performing the experiment with large elements dramatically brought down the speed of the vector. The vector significantly underperformed compared to the linked list, taking about 12 times longer than the linked list for 12,000 large containers.




Vector VS List — Large Container, Seed: 1
Microseconds
3.5×10^7
3.0×10^7
2.5×10^7
2.0×10^7
1.5×10^7
1.0×10^7
5×10^6
0
0
2×10^3
4×10^3
6×10^3
8×10^3
1.0×10^4
1.2×10^4
Number of Elements


Vector VS List — Large Container, Seed: 2
Microseconds
3.5×10^7
3.0×10^7
2.5×10^7
2.0×10^7
1.5×10^7
1.0×10^7
5×10^6
0
0
2×10^3
4×10^3
6×10^3
8×10^3
1.0×10^4
1.2×10^4
Number of Elements

Cache usage
The vector was clearly less efficient than the linked list. However, it's not clear if that's because of cache misses or simply a larger amount of elements being shifted per insertion. To shed light on this, I ran a single trial inserting and deleting 10,000 large containers from a vector and then from a linked list. Instead of using 1,000 byte containers, I used 200 byte containers (Valgrind would have taken too long otherwise).

Vector Cache Report – Large Container (200 bytes)

```
==16202==
==16202== Events    : Ir Dr Dw I1mr D1mr D1mw ILmr DLmr DLmw
==16202== Collected : 6552524041 3862962552 3588733735 2437 184560403 372127 2355 74249 202658
==16202==
==16202== I   refs:      6,552,524,041
==16202== I1  misses:            2,437
==16202== LLi misses:            2,355
==16202== I1  miss rate:          0.0%
==16202== LLi miss rate:          0.0%
==16202==
==16202== D   refs:      7,451,696,287  (3,862,962,552 rd + 3,588,733,735 wr)
==16202== D1  misses:      184,932,530  (  184,560,403 rd +       372,127 wr)
==16202== LLd misses:          276,907  (       74,249 rd +       202,658 wr)
==16202== D1  miss rate:           2.4% (          4.7%   +           0.0%  )
==16202== LLd miss rate:           0.0% (          0.0%   +           0.0%  )
==16202==
==16202== LL refs:         184,934,967  (  184,562,840 rd +       372,127 wr)
==16202== LL misses:           279,262  (       76,604 rd +       202,658 wr)
==16202== LL miss rate:            0.0% (          0.0%   +           0.0%  )
```

List Cache Report – Large Container (200 bytes)

```
==16805==
==16805== Events    : Ir Dr Dw I1mr D1mr D1mw ILmr DLmr DLmw
==16805== Collected : 2414031451 880957400 557036050 2359 52651078 253337 2286 71451 152282
==16805==
==16805== I   refs:      2,414,031,451
==16805== I1  misses:            2,359
==16805== LLi misses:            2,286
==16805== I1  miss rate:          0.0%
==16805== LLi miss rate:          0.0%
==16805==
==16805== D   refs:      1,437,993,450  (880,957,400 rd + 557,036,050 wr)
==16805== D1  misses:       52,904,415  ( 52,651,078 rd +     253,337 wr)
==16805== LLd misses:          223,733  (     71,451 rd +     152,282 wr)
==16805== D1  miss rate:           3.6% (        5.9%   +         0.0%  )
==16805== LLd miss rate:           0.0% (        0.0%   +         0.0%  )
==16805==
==16805== LL refs:          52,906,774  ( 52,653,437 rd +     253,337 wr)
==16805== LL misses:           226,019  (     73,737 rd +     152,282 wr)
==16805== LL miss rate:            0.0% (        0.0%   +         0.0%  )
```

The results of these measurements are very telling. The linked list's performance remained on a similar scale, only increasing in cache misses by 10,000,000. Note that this result could be influenced by the need to read and write many more bytes in the process of storing the arrays contained within the large_container class.

As for the vector, it required approximately 7.5 billion memory reads and writes, compared to the list's 1.5 billion. In the previous cache experiment (10,000 integers), the vector had required only 1.2 billion memory reads and writes. This makes sense, since the vector now needs to shift around 200 times as many bytes per insertion. The most drastic change, however, is in the amount of cache misses for

the vector – 184 million, up from 675 thousand in the first experiment. That's about 270 times the cache misses! So, a potent combination of cache misses and increased reads/writes cripple the performance of the vector for large containers.

Conclusion

For the initial setup – storing and deleting small integers – the list performed much worse than the vector because it was **maximizing cache misses** during traversal. On the other hand, the vector was benefitting from cache pre-fetching due to its contiguous storage of elements. Even though the vector needed to shift elements around during insertion, most of the operations were being performed in cache.

The tables turned when I performed the experiment with 1,000 byte containers instead of integers. The list outperformed the vector due to two factors: firstly, the vector was shifting more bytes for each insertion than it had for the first experiment, while the linked list was still just adjusting a few pointers for each insertion. Secondly, the size of the vector prevented it from fitting in cache, resulting in many more cache misses.

Therefore, in situations where small elements are being stored, vectors will perform better than linked lists. However, if large elements need to be stored, it makes more sense to use a linked list instead of a vector.