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.

**Experiment #1 - Approach**

As specified by Professor Stroustrup, I timed the insertion of random elements into the list, as well as the deletion of 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, and 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 the insertion and deletion various amounts of elements. I started from 0 elements and worked my way up to 20,000, taking steps of 1,000.

**Inserting elements**

Since the assignment specifications called for the same algorithm for both linked lists and vectors, I decided to use an iterative method for insertion.

**Experiment #1 - Hypothesis**

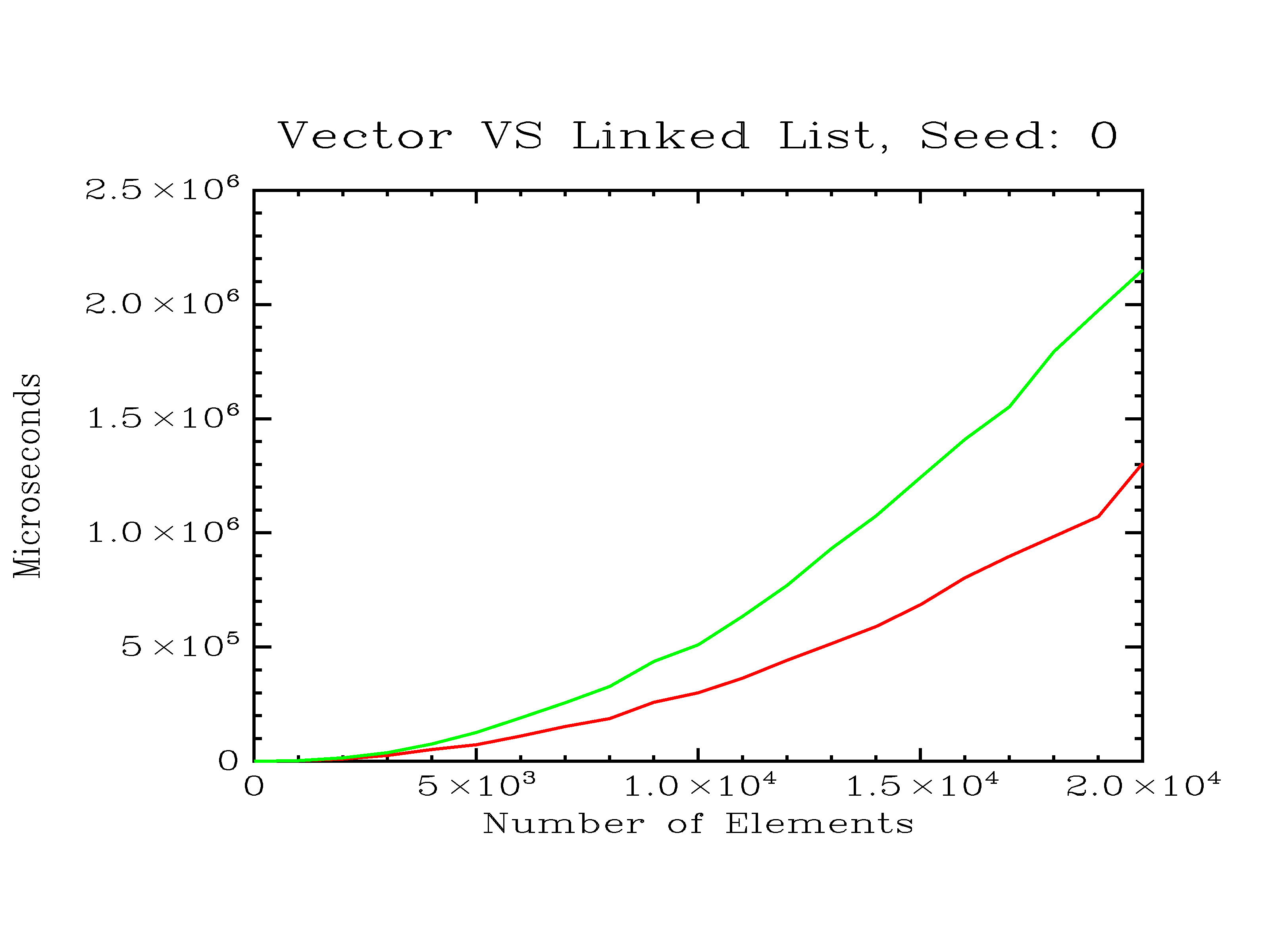
When I started this assignment, I simply assumed that the vectors would outperform the linked lists for a low number of elements. Then, at some point, the linked lists would start outperforming the vector. My logic was as follows: vectors incur an O(n) cost for insertions, since they store underlying elements in arrays. When an element is inserted into its appropriate position, all 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.

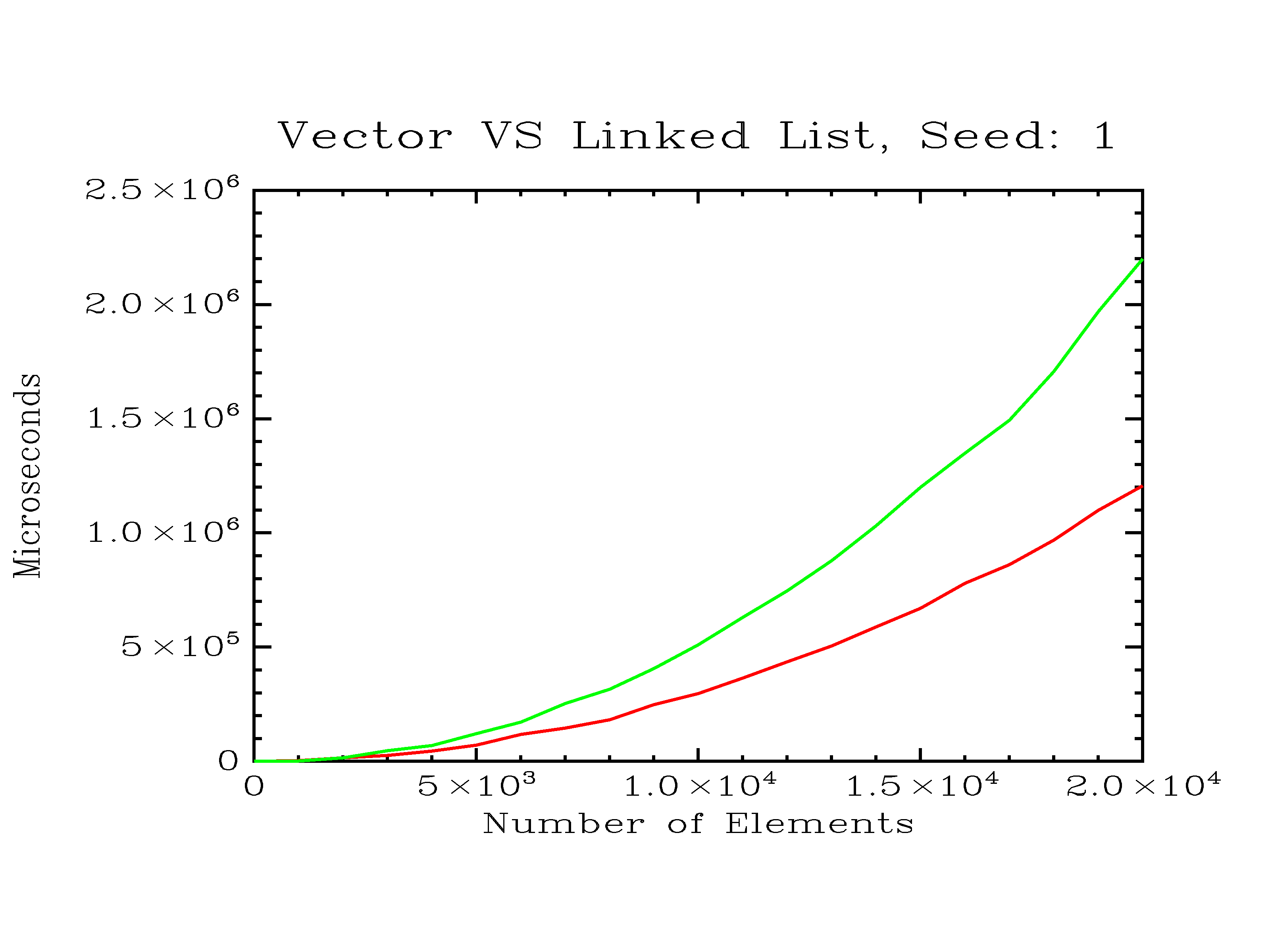
Linked lists, on the other hand, perform insertions in constant time – assuming that an iterator that points to the desired insertion location is readily available. Thus, for large n, the vector’s insertion time should grow linearly, while the linked list’s remains the same.

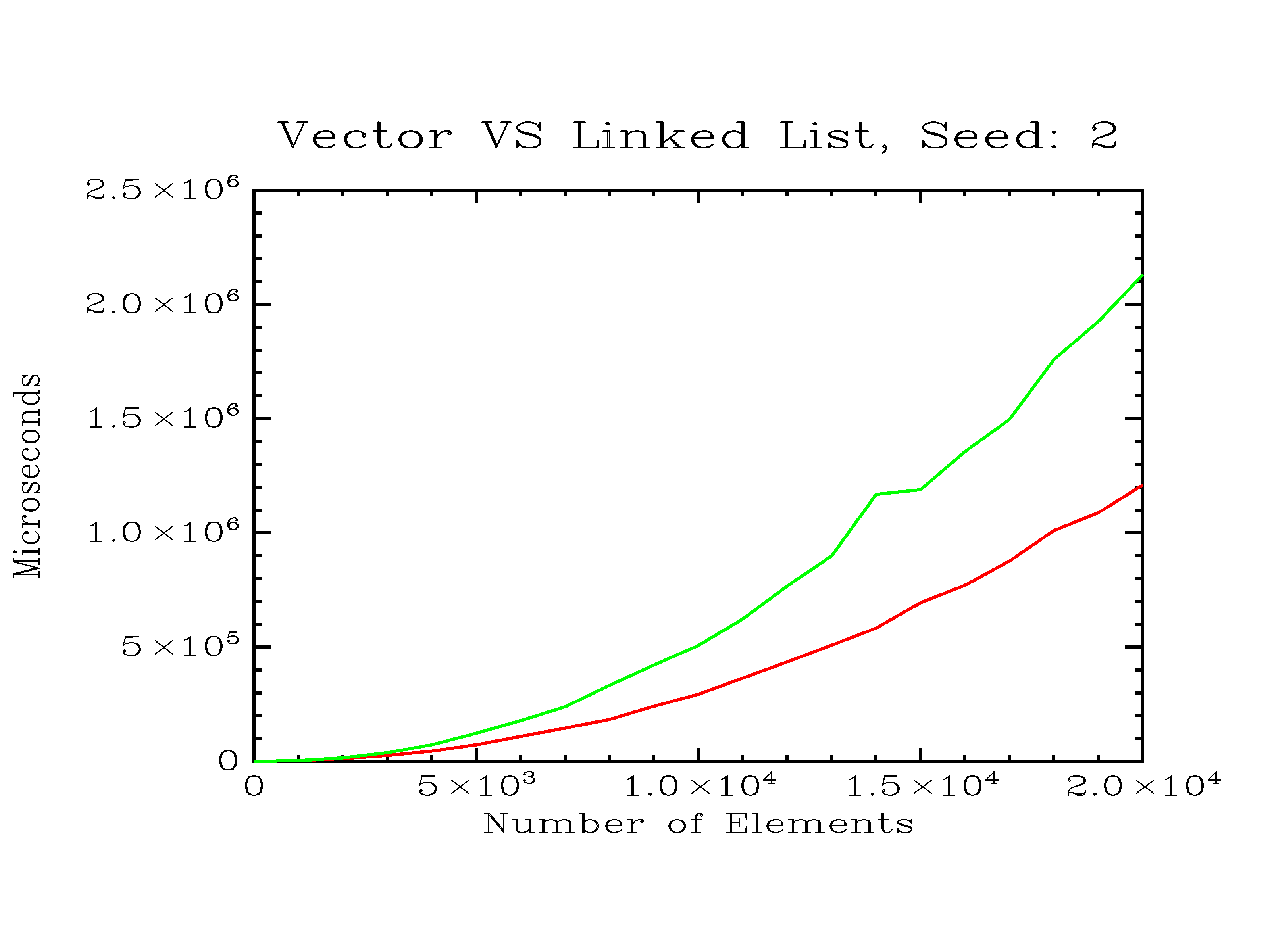
In total, the time complexity for vectors should be O(n3), dominated by element insertion. This is because for each element, an O(n) traversal is performed, followed by an O(n) insertion. For linked lists, the time complexity should be O(n2). For each element, and O(n) traversal is performed, followed by an O(1) insertion. Because of this difference in complexities, the linked list should eventually outperform the vector.

**Experiment #1 - Results**

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







**Experiment #1 - Analysis**

The vector’s superiority over the linked list must be somehow accounted for. After discussing my 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, all of the shifting that occurs during insertion happens within the cache.

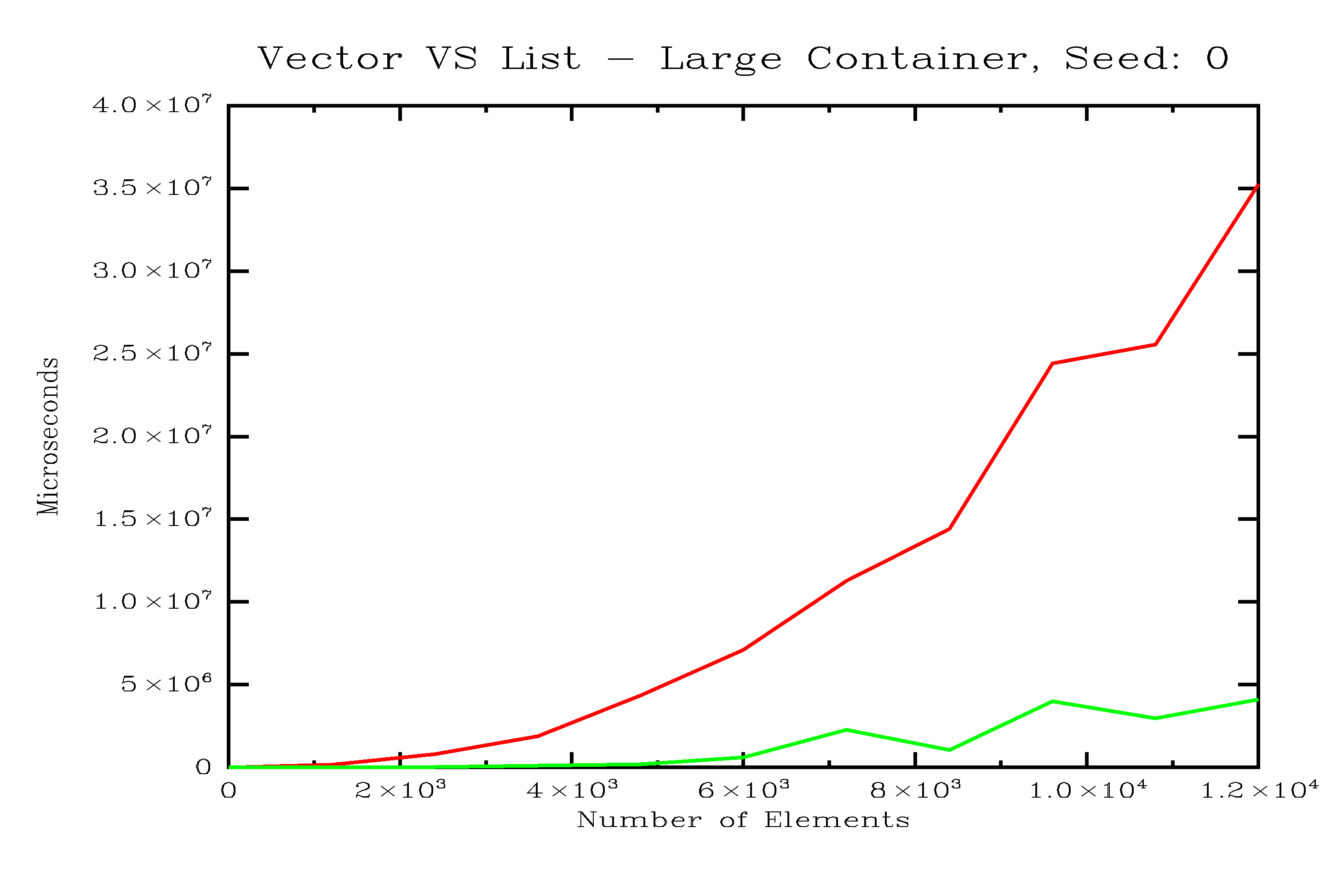
For linked lists, on the other hand, all of the data is scattered across memory instead of getting stored contiguously. Thus, the cache cannot help the list. For this hypothesis to be true, we should be able to bring down the speed of the vector by storing large pieces of data – so large that a relatively small amount of them will exhaust the size of the cache.

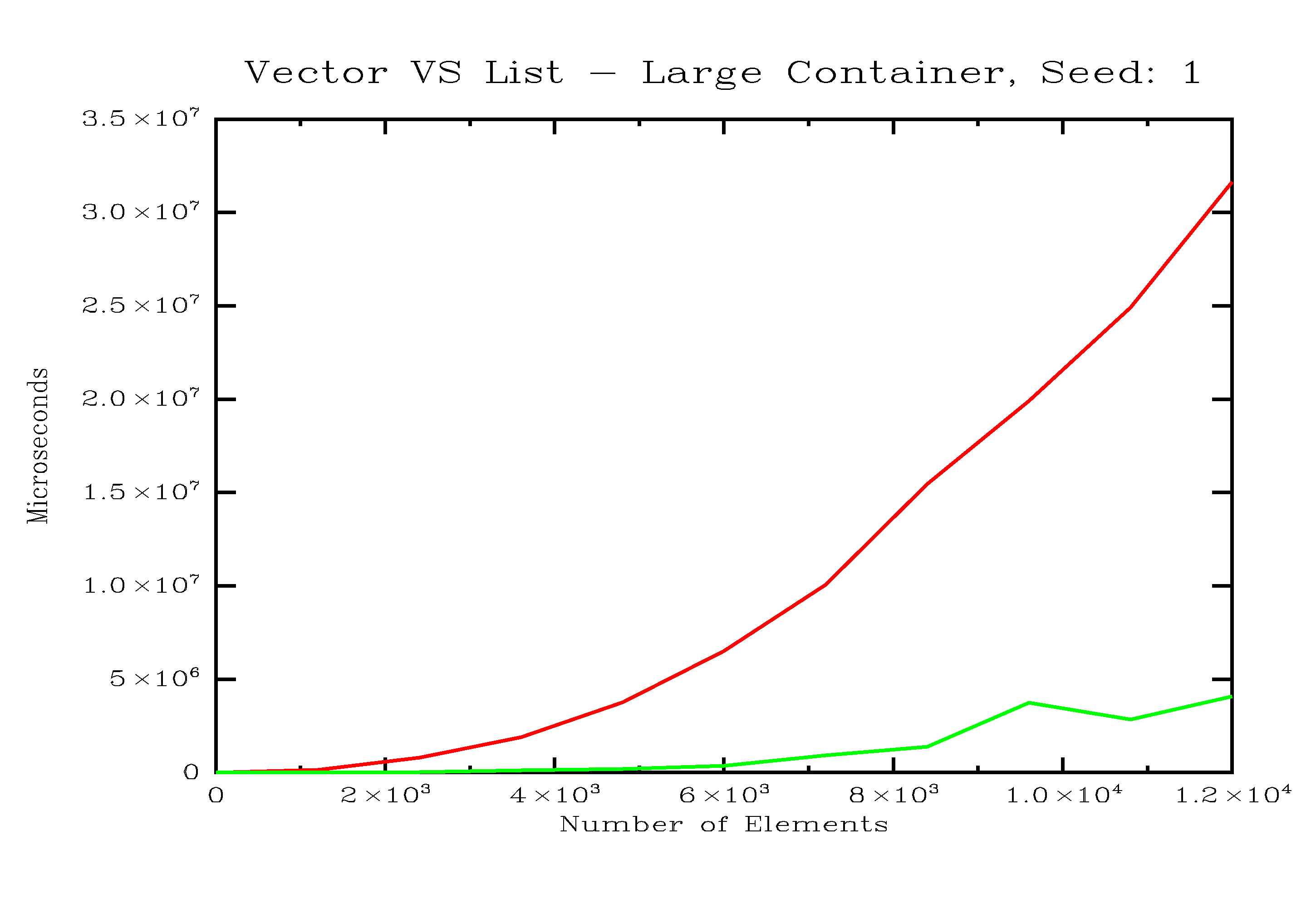
**Experiment #2 – Method & Hypothesis**

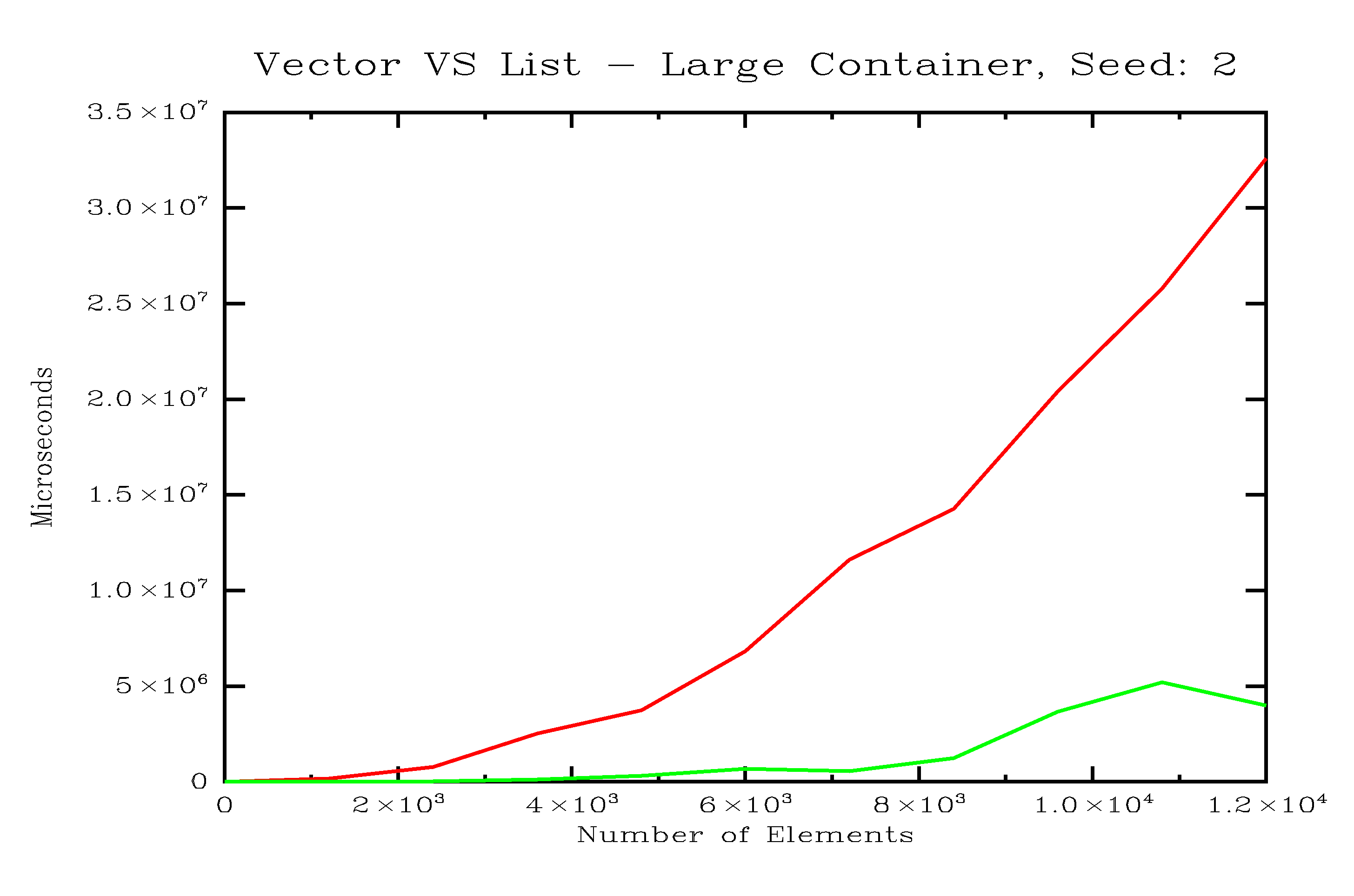
To test my claim, I performed the same experiment, but instead of inserting and deleting integers, I inserted and deleted elements of my custom class, **large\_container.** Large container simply contains an integer as a private member, as well as a 1000-byte char array. The idea is that this array will take up enough space to prevent the pre-fetching mechanism from storing a sufficient amount of elements in the cache.

**Experiment #2 – Results**

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

****

****

****

**Experiment #2 – Analysis**

In the original test, which involved the insertion and deletion of integers into linked lists and vectors, the vector always outperformed the linked list. However, when the experiment changed and involved the insertion and deletion of byte-sized elements, the vector’s speed worsened dramatically.

This leads us to conclude that for small elements, vectors benefit enormously from cache. For large elements, the size of the cache is too small to store enough of these elements to benefit the vector.

**Experiment #3 – Method**

One aspect that I felt was missing from the prior two experiments was determining at which point the cache can no longer store enough elements to help the vector. In the experiment with **large\_container**, the element size was upped to 1000 bytes. Of course, this would create difficulties for vectors, because it forces the vectors to shift over 1000 times the amount of bytes for each insertion. That is, it is not at all surprising that an increased element size would cause the vector to perform worse – regardless of the cache. So, it makes sense to run an experiment that determines where exactly the cache fails.

To figure out where this point is, I decreased the container size from 1000 bytes to 70 bytes, and performed the same experiment.

**Experiment #3 – Hypothesis**

**Conclusion**

For the initial setup – storing and deleting small integers – the list’s runtime was dominated by **traversing the list**. 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.**