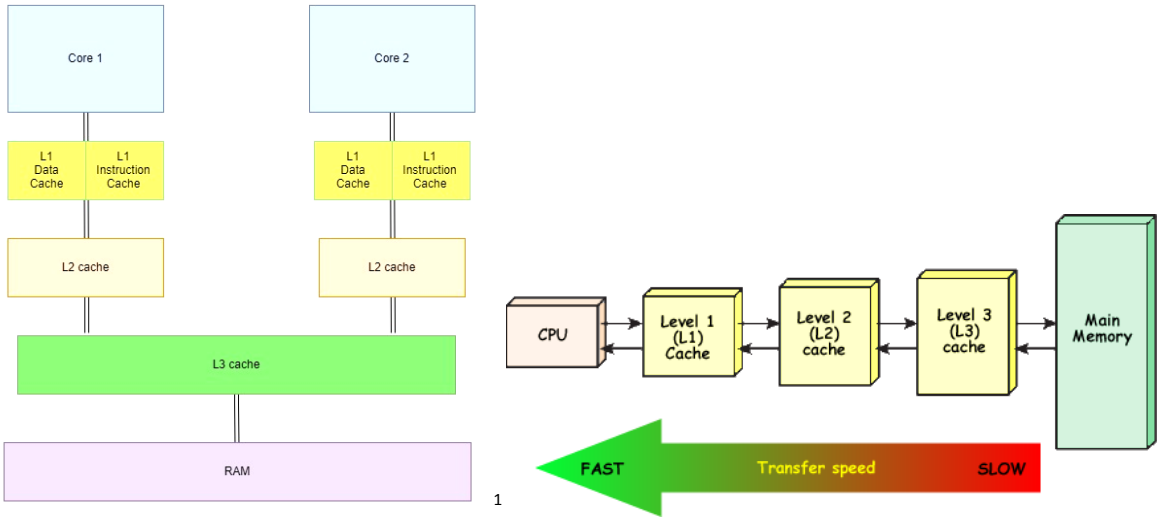
Chapter 3 Part 2 Supplement: Linked List

Understanding the Hidden Cost of Data Structures

By now you know that traversing all the elements of a linear array and doing that on a linked list have the same computational cost if the number of elements in the array and the list are the same. The cost is some multiple of the number of elements, N. However, what multiple? That is a big question and in a real world setting that matters a lot. To understand the issue, you need a little bit of initiation on how the computer CPU and memory are organized.

From the early days of computers, CPU is many times faster than the RAM (which is the memory) which in turn many times faster than the DISK (which is the computer's data storage). Since, to do any computation in the CPU, you need data; the computer designers applied a simple optimization to avoid hurting computer's performance due to the slowness of data read/write. The idea is that the RAM loads a large block of data at once, which is several KB or MB, from the storage even if the CPU asks for a single byte of data. The reason is, we expect in any typical program, CPU will ask for more data from locations nearby to its first request.

Surprisingly, or not, this assumption happens to be true for more than 98% of time for all data accesses of a program. However, even with this strategy, hardware designer realized that the slowness of the RAM (no longer the slowness of the disk, which has been taken care of) hurts overall program performance greatly. So they just reused the same idea by introducing something called caches. Caches are like RAM but much smaller and faster (their small size is related to their faster speed).



[1]

[1] Images: https://www.cybercomputing.co.uk/Languages/Hardware/cacheCPU.html, https://miro.medium.com/max/1030/1*Do424lmt3V06kJcmU1CEgg.png

So when a program runs, if the CPU ask for a single byte of data that is available in the RAM, the hardware actually load multiple consecutive bytes, at the range of 256 bytes or more from the RAM to the cache. The reason is the same; the hardware hopes that the program will make request for data among the additional bytes in near future. At that time, the request can be served from the cache. This idea of loading more data in closer memory works so good that now a days, all computers has multiple levels of caches between the CPU and the RAM.

This strategy for optimization affects the performance of element traversals of an array and a list differently. As you know, array elements are allocated in consecutive memory cells during the creation of the array. Therefore, traversal is extremely fast. There are just occasional data transfers between the cache and the RAM. However, space for a list element is allocated when you add the element to the list. The hardware gives whatever the next free memory cells available at that time. As a result, the elements are actually in different places of the RAM and are only connected together by the links of the linked list. So when you traverse a linked list, the hardware has to fetch data from the RAM to caches many times more. Therefore, traversing a linked list is significantly slower than traversing a linear array of the same size. It is like using Google MAP to go from a source location to a destination location. It asks you whether you will walk, bike (bicycle), take a car, or a bus despite the routes (aka intermediate locations you need to cross) being the same. Traversing the array is like taking a bus or a car. Traversing a linked list is like walking or biking.

Why you do not see the performance difference in your assignment programs? The reason it, the arrays and lists you use in your assignments are so small that both typically fit within the amount of data hardware load from the RAM to the cache in a few reads.