For the greatest size, 80,000 elements, the most time efficient processes tend to be fillArrayByAppend and fillListByAppend. This is because appending a value is an efficient operation in that it involves neither shifting nor searching; the new element goes directly into the next available position in the array or at the end of the linked list. Generally the slowest processes are fillArrayInOrder and fillListInOrder. These processes maintain an ordered structure where elements have to be shifted in the case of arrays or traversals done in the linked list to find an appropriate insertion point. Hence, time complexity increases. Arrays suffer further in this case, as it requires contiguous memory, and for that purpose, reallocation of memory is necessary. In the case of linked lists, dynamic memory allocation can be used, but again due to traversals, delays take place.

This gap increases with size: in particular, in-order insertion processes start to be relatively slower compared to the appending or prepending methods because of the shifting cost in arrays or node traversal in linked lists, which increases with the size of the structure. The array-based methods tend to have a larger slowdown due to the need to keep contiguous memory and potentially resize, which introduces added overhead. For the linked lists, while their memory allocation might handle size increases more gracefully, the traversal time still grows linearly with the number of elements, so the insertion operations are bound to become slower as the size increases.

Arrays and linked lists each have different strengths depending on usage. Of course, arrays exhibit faster access times due to indexed searches being in O(1), and of course, represent a more memory-efficient option when the size can be predecided. Also, on the other hand, when a large number of insertions and deletions are about to be done, linked lists would instead be more favorable as such operations could be made in constant time-O(1)-provided that this happens in the front or the end of the queue. The scenario where it is worth using a doubly linked list includes all the situations when one might want to traverse a list in both directions, for example, navigation systems. A cost of this is that, because of the additional pointers required, more memory is used by a doubly linked list. A circularly linked list is especially useful for representing a cyclical data structure-for example, round-robin scheduling, since no special case checking for the end of the list is required.