Assesment : Runtime.js comments

Ran the test first time:
=======================

Results for the tinyArray
insert 77.615 μs
append 187.808 μs

Results for the smallArray
insert 84.278 μs
append 187.886 μs

Results for the mediumArray
insert 303.499 μs
append 363.678 μs

Results for the largeArray
insert 16.250241 ms
append 1.081365 ms

Results for the extraLargeArray
insert 2.718046379 s
append 7.889804 ms

Ran the test a second time:
========================

Results for the tinyArray
insert 73.659 μs
append 437.709 μs

Results for the smallArray
insert 22.987 μs
append 28.637 μs

Results for the mediumArray
insert 218.833 μs
append 72.832 μs

Results for the largeArray
insert 23.350088 ms

Assesment : Runtime.js comments

append 1.08041 ms

Results for the extraLargeArray
insert 2.726735278 s
append 75.562086 ms

Conclusion:

The “doubleInsert " function seems to have better time performance for small array sizes in terms of microseconds but, when size increases it degrades in time performance. It could be because of the increase in quadratic shifting operation as the array size grows.
On the other hand, “doublerAppend performs much faster comparatively when the size increases..
As the size of Array increases the “doubleAppend” performs much better and the doubleInsert function’s performance degrades more rapidly, indicating that it does not scale well compared to the doublerAppend function.
So, doublerAppend function is likely to be more efficient and scalable.

Research:

The slower function (doublerinsert), which uses the unshift method for array insertion (doublerInsert), exhibits significantly slower performance compared to the faster function that uses the push method for array insertion (doublerAppend). This difference in performance is primarily attributed to how these array insertion methods operate and their impact on memory management.

1. Insertion Mechanism:

push (doublerAppend): The push method appends elements to the end of an array. It involves minimal overhead, especially for dynamic arrays, as elements are added at the array's current end.
unshift (doublerInsert): The unshift method inserts elements at the beginning of an array. This requires shifting existing elements to make room for the new element, which becomes increasingly expensive as the array grows.
2. Memory Shifting:

push: When appending elements at the end of an array, memory allocation and copying are typically less intensive, as new elements are placed in contiguous memory locations.
unshift: Inserting elements at the beginning of an array requires shifting the entire array's contents to make space for the new element. This process involves copying each element to the next position, resulting in multiple memory copy operations. As the array size grows, this shifting becomes a performance bottleneck.
3. Time Complexity:

Assesment : Runtime.js comments

push: The push method generally has time complexity (O(n) , making it efficient even for large arrays.
unshift: The unshift method has a linear time complexity (O(n^2)), where n is the number of elements in the array. This complexity arises from the need to shift all existing elements by one position.
4. CPU Cache Effects:

When shifting elements using unshift, there is a higher likelihood of cache misses. Cache misses occur when the CPU has to retrieve data from main memory instead of the faster cache memory. This can lead to increased memory access times and slower execution.
5. Memory Allocation:

The dynamic array used by JavaScript may need to be resized when elements are inserted at the beginning. Resizing involves allocating new memory, copying existing elements, and deallocating old memory, adding additional overhead to the unshift operation.
6. Engine Optimizations:

JavaScript engines may apply various optimizations, such as avoiding memory shifts for small arrays or preallocating space, to mitigate the performance impact of unshift. These optimizations can lead to variations in observed execution times based on the specific engine and environment.
In summary, the unshift method's slower performance can be attributed to its need to shift existing elements, its O(n^2) time complexity, potential cache effects, and memory management overhead. As the array size increases, the cost of these operations becomes more pronounced, causing the unshift-based function to scale poorly compared to the push-based function, which appends elements at the end.