TIME COMPLEXITY ANALYSIS

METHOD	SINGLY LINKED LIST	DYNAMIC ARRAY
Insert at index	O(N)	O(N)
Delete at index	O(N)	O(N)
Returns size	O(N)	O(1)
Empty or not	O(1)	O(1)
Rotate	O(n+k)	O(k)
Reverse	O(N)	O(N)
Append at End	O(N)	O(N)
Prepends at Beginning	O(1)	O(N)
Merge	O(n+m)	O(n+m)
Interleaves	O(n+m)	O(n+m)
Get middle element	O(N)	O(1)
Index of element	O(N)	O(N)
Splits at Index	O(N)	O(N)

SPACE COMPLEXITY ANALYSIS

OPERATION	SINGLY LINKED LIST	DYNAMIC ARRAY
Memory Overhead	O(N)	O(N)
Space Utilization	May have unused space due to overallocatio	No unused space beyond node pointers

DYNAMIC ARRAYS

Advantages:

- **Fast Access:** Direct Access to any element by index in constant time.
- **Cache-Friendly:** Elements are stored in contiguous memory locations which makes them cache-friendly.
- **Dynamic Resizing:** Can dynamically resize to accommodate more elements, as seen with the '**__resize**' method.

Disadvantages:

- **Expensive Inserts/Deletes:** Inserting or deleting elements, especially at the beginning or in the middle, is costly as it requires shifting elements.
- **Resizing Overhead:** Occasionally, resizing the array (usually doubling its size) can be an expensive operation.

SINGLY LINKED LIST

Advantages:

- **Efficient Insertion/Deletions:** Insertions and deletions at the beginning are very efficient, O(1) time complexity.
- **Dynamic Size:** Can grow or shrink in size easily without requiring reallocation or resizing.
- **Flexible Node Management:** Merging, Interleaving, and splitting operations are straightforward and efficient.

Disadvantages:

- **Slow Access:** Accessing elements by index is slow since it requires traversing the list from the head.
- **Higher Memory Overhead:** Requires extra money for pointers which can add up, especially if the data stored is small.
- **Cache Inefficiency:** Nodes are not stored contiguously in memory, which can result in poor cache performance.

DETAILED COMPARISON

Dynamic Array Methods:

- **Insert at index:** O(n) due to shifting elements.
- **Delete at index:** O(n) due to shifting elements.
- **Rotate:** O(k) for rotating the array.
- **Reverse:** O(n) for reversing the array.
- **Merge:** O(n + m) for combining two arrays.
- **Interleave:** O(n + m) for interleaving two arrays.
- **Get middle element:** O(1) as it accesses the middle directly.

- **Index of element:** O(n) as it searches through the array.
- **Split at index:** O(n) for creating two new arrays.

Singly Linked List Methods:

- **Insert at index:** O(n) due to traversing the list.
- **Delete at index:** O(n) due to traversing the list.
- **Rotate:** O(n + k) since it requires finding the new head and re-linking nodes.
- **Reverse:** O(n) for reversing the links.
- **Merge:** O(n + m) for combining two lists.
- **Interleave:** O(n + m) for interleaving two lists.
- **Get middle element:** O(n) due to traversing to the middle.
- **Index of element:** O(n) as it searches through the list.
- **Split at index:** O(n) for traversing and splitting the list.

Conclusion:

Both data structures have unique advantages and disadvantages, making them suitable for different scenarios:

- **Dynamic Arrays** are ideal for scenarios requiring fast access by index and occasional insertions/deletions, primarily at the end.
- **Singly Linked Lists** excel in scenarios with frequent insertions/deletions, especially at the beginning, and when dynamic resizing is needed without overallocation.

Choosing the right data structure depends on the specific requirements of your application, such as the frequency of insertions, deletions, and access operations. For instance, if you need constant-time access and are willing to tolerate occasional resizing costs, a dynamic array is preferable. On the contrary, if you need efficient insertions and deletions and can tolerate slower access times, a singly linked list is a better choice.