Here's a **comprehensive, big table** incorporating **all methods** from your updated reference table, organized neatly for clarity. I've color-coded sections for better readability and added **key notes** to help you learn faster.

Java List Implementations: Complete Method Comparison
Legend:
🟢 = Common Methods | 🔵 = Deque Methods (LinkedList) | 🟡 = Stack Methods | 🟣 = CopyOnWriteArrayList-Specific

Method	ArrayList	LinkedList	Vector	Stack	CopyOnWriteArrayList	Inherited From
🟢 **Basic List/Collection Methods**						
`boolean add(E e)`	Yes	Yes	Yes	Yes	Yes	`Collection`
`void add(int index, E element)`	Yes	Yes	Yes	Yes	Yes	`List`
`boolean addAll(Collection c)`	Yes	Yes	Yes	Yes	Yes	`Collection`
`boolean addAll(int index, Collection c)`	Yes	Yes	Yes	Yes	Yes	`List`
`void clear()`	Yes	Yes	Yes	Yes	Yes	`Collection`
`boolean contains(Object o)`	Yes	Yes	Yes	Yes	Yes	`Collection`
`boolean containsAll(Collection c)`	Yes	Yes	Yes	Yes	Yes	`Collection`
`E get(int index)`	Yes	Yes	Yes	Yes	Yes	`List`
`int indexOf(Object o)`	Yes	Yes	Yes	Yes	Yes	`List`
`boolean isEmpty()`	Yes	Yes	Yes	Yes	Yes	`Collection`
`Iterator<E> iterator()`	Yes	Yes	Yes	Yes	Yes	`Collection`

`int lastIndexOf(Object o)`	Yes	Yes	Yes	Yes	Yes	`List`
`ListIterator<E> listIterator()`	Yes	Yes	Yes	Yes	Yes	`List`
`ListIterator<E> listIterator(int)`	Yes	Yes	Yes	Yes	Yes	`List`
`E remove(int index)`	Yes	Yes	Yes	Yes	Yes	`List`
`boolean remove(Object o)`	Yes	Yes	Yes	Yes	Yes	`Collection`
`boolean removeAll(Collection c)`	Yes	Yes	Yes	Yes	Yes	`Collection`
`boolean retainAll(Collection c)`	Yes	Yes	Yes	Yes	Yes	`Collection`
`E set(int index, E element)`	Yes	Yes	Yes	Yes	Yes	`List`
`int size()`	Yes	Yes	Yes	Yes	Yes	`Collection`
`List<E> subList(int, int)`	Yes	Yes	Yes	Yes	Yes	`List`
`Object[] toArray()`	Yes	Yes	Yes	Yes	Yes	`Collection`
`<T> T[] toArray(T[] a)`	Yes	Yes	Yes	Yes	Yes	`Collection`
🟤 **Stack-Specific Methods**						
`void push(E e)`	No	No	No	Yes	No	`Stack`
`E pop()`	No	No	No	Yes	No	`Stack`
`E peek()`	No	No	No	Yes	No	`Stack`
🟤 **Deque Methods (LinkedList)**						
`void addFirst(E e)`	No	Yes	No	No	No	`Deque`

`void addLast(E e)`	No	Yes	No	No	No	`Deque`
`E getFirst()`	No	Yes	No	No	No	`Deque`
`E getLast()`	No	Yes	No	No	No	`Deque`
`boolean offerFirst(E e)`	No	Yes	No	No	No	`Deque`
`boolean offerLast(E e)`	No	Yes	No	No	No	`Deque`
`E peekFirst()`	No	Yes	No	No	No	`Deque`
`E peekLast()`	No	Yes	No	No	No	`Deque`
`E pollFirst()`	No	Yes	No	No	No	`Deque`
`E pollLast()`	No	Yes	No	No	No	`Deque`
`E removeFirst()`	No	Yes	No	No	No	`Deque`
`E removeLast()`	No	Yes	No	No	No	`Deque`
🔘 **CopyOnWriteArrayList-Specific**						
`boolean addIfAbsent(E e)`	No	No	No	No	Yes	`CopyOnWriteArrayList`
`int addAllAbsent(Collection c)`	No	No	No	No	Yes	`CopyOnWriteArrayList`

Key Notes for Quick Learning

1. **Thread Safety**:
    a. 🔘 **`Vector/Stack`**: Legacy synchronized classes (slow in multi-threading).
    b. 🔘 **`CopyOnWriteArrayList`**: Thread-safe with snapshot iterators (no locks for reads).
2. **Performance**:

    a. **`ArrayList`**: Fast random access (`O(1)`), slow inserts/deletes in the middle (`O(n)`).
    b. **`LinkedList`**: Fast inserts/deletes at ends (`O(1)`), slow random access (`O(n)`).
    c. **`CopyOnWriteArrayList`**: Expensive writes (copies entire array), ideal for read-heavy concurrency.
3. **When to Use**:
    a. **`ArrayList`**: Default for most use cases.
    b. **`LinkedList`**: Queues, stacks, or frequent insertions/deletions.
    c. **`CopyOnWriteArrayList`**: Thread-safe listener lists or observer patterns.
    d. **Avoid `Vector`/`Stack`**: Use `Collections.synchronizedList()` or `ArrayDeque` instead.
4. **Legacy Alert**:
    a. `Vector` and `Stack` are deprecated. Use modern alternatives like `ArrayList` or `ConcurrentHashMap`.

Formatting Tips for Word

- **Colors**: Use the legend emojis (🟩, 🟨, 🟦, 🟥) to highlight sections.
- **Borders**: Add light gray borders to separate rows/columns.
- **Font**: Use **monospace** for method names (e.g., `add()`) for clarity.