1a)
This is helpful because adding or removing to the end of the list becomes much easier and quicker. There is no longer a need to iterate through the whole list just to add or remove an element to the end.

1b)

Method	ArrayList	LinkedList
add(int value)	O(1)	O(1)
add(int index, int value)	O(n)	O(n)
clear()	O(1)	O(1)
contains(int value)	O(n)	O(n)
get(int index)	O(1)	O(n)
isEmpty()	O(1)	O(1)
remove(int index)	O(n)	O(n)
toString()	O(n)	O(n)
equals(Object o)	O(n)	O(n)

1c)
I would use an ArrayList over a LinkedList when i want to access elements from the middle of the list because getting an element from the middle of an ArrayList is faster than getting an element from the middle of a LinkedList

1d)
I would use a LinkedList over an ArrayList when I want to add elements to the front of the list because adding an element to the front of a LinkedList is much easier than doing the same to an ArrayList.

2a)

Method	ArrayStack	ListStack
push(int value)	O(n)	O(1)
pop()	O(n)	O(1)
peek()	O(1)	O(1)
isEmpty()	O(1)	O(1)
size()	O(1)	O(1)
clear()	O(1)	O(1)
toString()	O(n)	O(n)
equals(Object o)	O(n)	O(n)

2b)
I would choose the linked list stack because the push and pop have faster runtime and all the other methods have the same runtime

3a)

Method	ArrayQueue	ListQueue
enqueue(int value)	O(n)	O(1)
dequeue()	O(n)	O(1)
peek()	O(1)	O(1)
isEmpty()	O(1)	O(1)
size()	O(1)	O(1)
clear()	O(1)	O(1)
toString()	O(n)	O(n)
equals(Object o)	O(n)	O(n)

3b)

I would choose the linked list queue for the same reasons as the linked list stack. Push and pop run faster and all the other methods run at the same runtime.