1. ArrayList vs LinkedList
    a. This would be helpful because we would have instant access to the back of the linked list as well as the front. This would help significantly with the peek and pop methods from the Stack implementation.
    b. Table

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

    c. A scenario you would want to choose an array list is when you need fast random access to the data. If you expect to be searching for things by index, you want an array list. The main benefit of the Linked List comes if you plan to traverse your list by iteration. When doing this, you can add or delete items in O(1) by using the .next pointers, while an array list has to shift an entire part of the array when performing these operations.
    d. An example of a linked list being the better choice would be a playlist queue. Each song object holds a pointer to the next song object, which can be changed if you want to queue up a different song next.
2. ArrayStack vs ListStack
    a. Table

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

   b. I would choose to use the array to implement the stack because the array has the lower O for 4 of the key functions of the class. Additionally, the linked list does not have any areas of performance benefit of all in this case.

3. ArrayQueue vs ListQueue
   a. Table

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

   b. I would use the linked list class because it allows for removal at very low performance cost. The fact it uses pointers allows for better performance;.