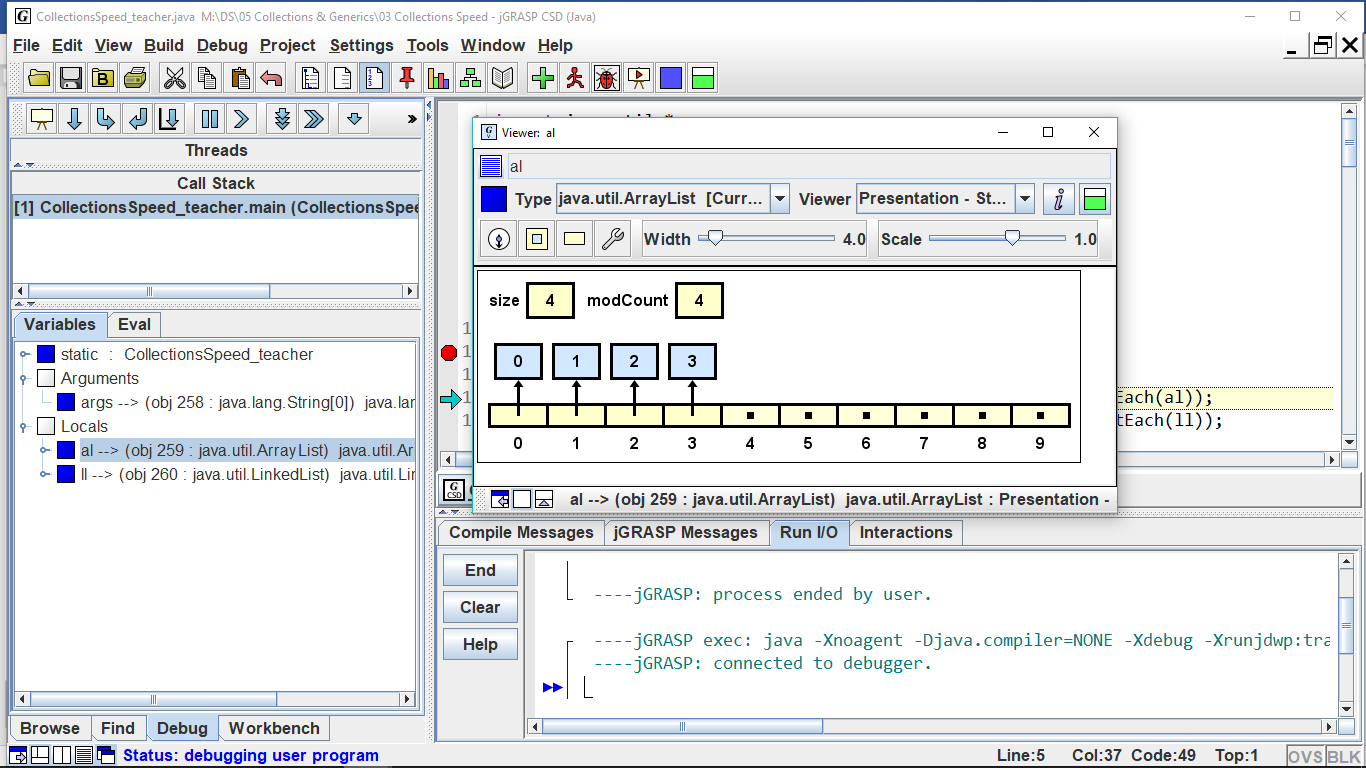
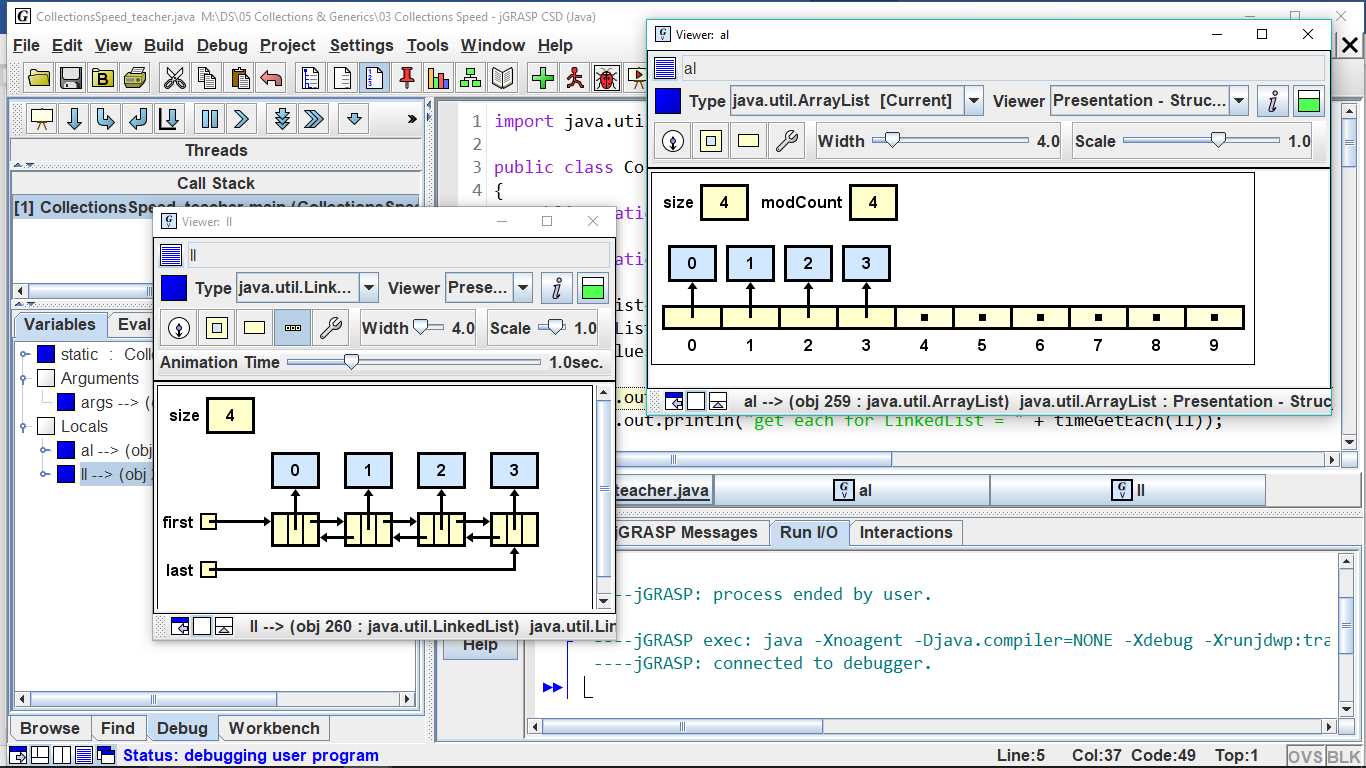
Big-O Values for LinkedList and ArrayList

As you have seen with TJArrayList, the java.util.ArrayList<E> maintains a backing "raw" array.



anArrayList

Similar to DLL, java.util.LinkedList<E> maintains a backing doubly linked list of list nodes with a pointer to the last node.



aLinkedList

|  |  |  |
| --- | --- | --- |
|  | **LinkedList** | **ArrayList** |
| size() |  |  |
| add(o) |  |  |
| add(i, o) |  |  |
| get(i) |  |  |
| set(i, o ) |  |  |
| remove(i) |  |  |
| addFirst(o) |  |  |
| addLast(o) |  |  |
| getFirst() |  |  |
| getLast() |  |  |
| removeFirst |  |  |
| removeLast() |  |  |

Because of their differing structure, the two implementations of List<E> have different Big-O performances. As you fill in the Big-O table, think about when you need to access each element, and when you can jump directly to the element.

Some of these methods are *index-based* and others are *object-based.*

Predict the Big-O values and explain why:

|  |  |  |  |
| --- | --- | --- | --- |
|  |  | if list is an ArrayList | If list is a LinkedList |
| 1. | for(int i=0; i<list.size(); i++)  total += list.get(i); |  |  |
| 2. | while( iter.hasNext() )  if(iter.next().equals("Bob"))  iter.remove(); |  |  |

Let's think about the Big-O for *ordered* (or sorted) ArrayLists and LinkedLists. "Ordered" means that every time you insert or delete, you must maintain the sorted order of the list.

Predict the Big-O values and explain why:

|  |  |  |  |
| --- | --- | --- | --- |
|  |  | if list is an ArrayList | If list is a LinkedList |
| 3. | Searching the list |  |  |
| 4. | Inserting an item into the list |  |  |
| 5. | Deleting an item |  |  |
| 6. | Insert an item that forces a resize |  |  |

When do you choose to use an ArrayList and when to use a LinkedList? Sometimes Big-O efficiencies make the difference.

1. If quick access to the data is important, use a(n) \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_
2. If quick insertion and deletion is important, use a(n) \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_
3. If the amount of data is relatively unchanging, use a(n) \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_
4. If the amount of data changes rapidly and widely, use a(n) \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_
5. If data is to be inserted at the front, use a(n) \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_because the Big-O is \_\_\_\_\_
6. If data is to be inserted at the mid-point, use \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_because the Big-O is \_\_\_\_\_
7. If data is to be inserted at the end, use \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ because the Big-O is \_\_\_\_\_\_
8. If you are accessing the data by its index, then a(n) \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ gives you an access time of O(1).
9. If you are accessing the data by its index, then a(n) \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ gives you an access time of O(n).