Complexity analysis of Bag Operations:
Arrays vs. Linked Lists

Why Do We Need Linked Lists?

- To overcome the disadvantages of using arrays:
  - **Fixed size:** to create an array you have to specify the size, however, a linked list can grow and shrink dynamically
  - **Adding at random positions:** adding an element to the front of an array (or in the middle) is very hard since a lot elements need to be copied to other locations in order to make space for the new element to be inserted
    - However, for a linked list, an element can be added at any location by performing a few assignment statements to adjust the links.

Guidelines for Choosing Between an Array and a Linked List

Operation	Which data structure to use?
Frequent random access operations	Array
Frequent Insertion and deletion at random locations	Linked list (to avoid moving elements up and down)
Frequent capacity change	Linked list (to avoid the resizing inefficiency)

Complexity of Adding to an Unordered Array-Based List

```
STEPS
0     public void add(int element)   {
1           data[manyItems] = element;
1          manyItems++;
      }
-------
Total:  2 (constant) = O(1)
```

O(n)

Complexity of Adding to an Unordered Linked List

```
STEPS
0     public void add(int element)     {
1           this.head = new IntNode(element,head);
1           this.manyNodes++;
      }
-------
Total:  2 (constant) = O(1)
```

O(n)

Complexity of Inserting into an Ordered Array-Based List

```
STEPS
0       public void insert(int num)      {
1               int i=0;
2*n             while (i < manyItems && (data[i] <= num))     {
1*n                     i++;
                }

1               for (int move = manyItems; move > i; move-- ) {
1                       data[move] = data[move-1];
                }

1               data[i] = num;
1               manyItems++;
        }
-----
Total:  1 + 3*n + 2 + 2 = 3*n + 5 = O(n)
```

O(n)

Complexity of Inserting into an Ordered Linked List

```
STEPS
0       public void insert (int newValue)       {
1               if (head == null)
1                       head = new IntNode(newValue, head);
1               else if (newValue < head.getData())
1                       head = new IntNode(newValue, head);
                else   {
??                      IntNode previousNode = findPrevious(newValue);
1                       previousNode.addNodeAfrer(newValue);
                }
        }
-------
Total:          2 OR 2 OR (?? + 1) = ??
```

findPrevious() Method for Ordered Lists

```
STEPS
0     public IntNode findPrevious(int newValue)   {
1           IntNode cur = head;
1*n         while (cur.getLink() != null &&
1*n               cur.getLink().getData() <
                  newValue)  {
1*n               cur = cur.getLink();
            }
1           return cur;
      }
------
Total:  3*n + 2 = O(n)
```

Complexity of Inserting into an Ordered Linked List

```
STEPS
0       public void insert (int newValue)       {
1               if (head == null)
1                       head = new IntNode(newValue, head);
1               else if (newValue < head.getData())
1                       head = new IntNode(newValue, head);
                else   {
3*n+2                   IntNode previousNode = findPrevious(newValue);
1                       previousNode.addNodeAfrer(newValue);
                }
        }
-------
Total:          2 OR 2 OR (3*n + 2 + 1) = 3*n + 3 = O(n) (worst case)
```

Complexity of Adding or Inserting into a List

add()/insert()	Unordered List	Ordered List
Array-based list	O(1)	O(n)
Linked list	O(1)	O(n)

Complexity of Removing from a List

remove()	Unordered List	Ordered List
Array-based list	O(n)	O(n)
Linked list	O(n)	O(n)

Removing from an array is O(n) because you need to search for the element to be removed. Once found, you can replace it with the last element in the array.

Time analysis for data structure operations

IntArrayBag

Operation	Time
`add` (without capacity increase)	O(1)
`add` (with capacity increase)	O(n)
`countOccurances`	O(n)
`remove`	O(n)
`getByIndex`	O(1)
`size`	O(1)

IntLinkedBag

Operation	Time
`add`	O(1)
`countOccurances`	O(n)
`listSearch`	O(n)
`listPosition`	O(n)