ADT List: supported operations

ADT List supports the following main operations

▪ Get element by position: `get(int index)`
▪ Search element: `indexOf(E element)`
▪ Add new element: `add(int index, E element)`
▪ Remove element by position: `remove(i)`

For some problems however these operations are insufficient and we need access to the underlying implementation of the data
Example: count occurrences

• Write a method \texttt{count} that counts the number of times a particular element \texttt{o} appears in a List:

```java
public static int count(List list, E o) {
    int counter = 0;
    for (int i=0; i<data.size(); i++) {
        E obj = data.get(i);
        if (obj.equals(o)) counter++;
    }
    return counter;
}
```

• \textbf{Question}: would this work well no matter if the List is an \textit{ArrayList} or a \textit{Linked List}?
Example: count occurrences

• Write a method `count` that counts the number of times a particular element `o` appears in a List:

```java
public static int count(List list, E o) {
    int counter = 0;
    for (int i=0; i<data.size(); i++) {
        E obj = data.get(i);
        if (obj.equals(o)) counter++;
    }
    return counter;
}
```

• **Answer:** No, this method is very inefficient for Linked Lists: `get(i)` always starts from the head and this is an O(n^2) loop
Efficient solutions are fundamentally different for:

- **Array List**
  ```java
  int count (E element){
      int counter = 0;
      for(int i=0; i<size; i++){
          if(data[i].equals(element))
              counter ++;
      }
      return counter;
  }
  ```

- **Linked List**
  ```java
  int count (E element){
      int counter = 0;
      Node finger = head;
      while(finger != null){
          if(finger.data.equals(element))
              counter ++;
          finger = finger.next;
      }
      return counter;
  }
  ```

- But the principle of ADT forbids the use of underlying data structures directly!
- We need a uniform interface to iterate over List elements efficiently
Efficient uniform iteration over List

• **Problem:** Efficient and uniform dispensing of values from the underlying data structures

• **Solution:** We create and use the common interface for iteration
Extending operations for List ADT

- get()
- indexOf()
- add()
- remove()
- size()
- isEmpty()
- clear()
- contains()

But also method for efficient data traversal
  - iterator()
**Iterator interface**

- Iterators provide support for efficiently visiting all elements of an underlying data structure
- We customize the implementation of the iterator depending on the data structure
- We abstract away the details of how to access elements

```java
public interface Iterator<E> {
    boolean hasNext() – are there more elements for iteration?
    E next() – return next element
}
```
Iterator for *ArrayList*

```java
public class ArrayListIterator implements Iterator{
    ArrayList list;
    int index;
    public ArrayListIterator (ArrayList list){
        this.list = list;
    }

    public boolean hasNext (){ 
        return (this.index < list.size());
    }

    public Object next(){
        return list.data[index++];
    }
}
```
Iterator for *Array List*

```java
public class ArrayListIterator implements Iterator{
    ArrayList list;
    int index;
    public ArrayListIterator (ArrayList list){
        this.list = list;
    }

    public boolean hasNext (){ return (this.index < list.size());
    }

    public Object next(){
        return list.data[index++];
    }
}
```
public class ArrayListIterator implements Iterator{
    ArrayList list;
    int index;
    public ArrayListIterator (ArrayList list){
        this.list = list;
    }

    public boolean hasNext (){  
        return (this.index < list.size());
    }

    public Object next(){
        return list.data[index++];
    }
}
public class ArrayListIterator implements Iterator{
    ArrayList list;
    int index;
    public ArrayListIterator(ArrayList list){
        this.list = list;
    }
    public boolean hasNext(){
        return (this.index < list.size());
    }
    public Object next(){
        return list.data[index++];
    }
}
Iterator for *Array List*

```java
public class ArrayListIterator implements Iterator{
    ArrayList list;
    int index;
    public ArrayListIterator (ArrayList list){
        this.list = list;
    }

    public boolean hasNext (){  
        return (this.index < list.size());
    }

    public Object next(){  
        return list.data[index++];
    }
}
```

Return the element at position *index*, and increment *index*.
ArrayList **iterator()** returns array-specific Iterator:

```java
public class ArrayList {
    Object[] data;
    int size;

    public Iterator iterator() {
        return new ArrayListIterator(this);
    }
}
```
public class LinkedListIterator implements Iterator{
    LinkedList list;  
    Node current;

    public LinkedListIterator (LinkedList list){
        this.list = list;
    }

    public boolean hasNext (){
    }

    public Object next(){
    }
}
public class LinkedListIterator implements Iterator{
    LinkedList list;
    Node current;

    public LinkedListIterator (LinkedList list){
        this.list = list;
    }

    public boolean hasNext (){}

    public Object next (){}
}
**Linked List** Iterator: `hasNext()`

Which of the following is the correct implementation of `hasNext()`?

A. `boolean hasNext(){
    return (this.list.size()>0)
}

B. `boolean hasNext(){
    return (current.next != null)
}

C. `boolean hasNext(){
    return (current!= null)
}

D. None of the above
**Linked List Iterator: next()**

Which of the following is the correct implementation of `next()`?

A. ```java
   Object next()
   {
       return this.list.get(current);
   }
```  

B. ```java
   Object next()
   {
       return this.list.get(current);
   }
```  

C. ```java
   Object next()
   {
       Object result = current.data;
       current = current.next();
       return result;
   }
```  

D. None of the above
public class LinkedListIterator implements Iterator{
    LinkedList list;
    Node current;
    ...
    public boolean hasNext (){ 
        return (current != null);
    }

    public Object next(){
        Object result = current.data;
        current = current.next;
        return result;
    }
}
Linked List with its own iterator

```java
public class LinkedList {
    Node head;
    int size;

    public Iterator iterator(){
        return new LinkedListIterator(this);
    }
}
```
Uniform Counting with iterator()
Works for both Array List and Linked List

```java
public int count (List list, Object o) {
    int counter = 0;
    Iterator iter = list.iterator();
    while (iter.hasNext())
        if(o.equals(iter.next())) counter++;
    return counter;
}
```
Iterators: notes

- Iterator objects provide a common interface for traversing List of values
- They capture the state of traversal
- They have access to internal data representations
- Always call hasNext() before calling next()!!!

- To implement iterator you need to understand the mechanics of the underlying data structure
- The traversal should be as fast as the underlying data structure permits