```java
import java.util.Iterator;
import java.util.NoSuchElementException;
/**
    The Linked List Class is the framework for a series of ListNodes
    that together function like a List. You must create a Linked List
    containing values all of a certain type. Values can be added and removed
    from a Linked list, among other functions. This Class also implements
    Stack, Queue, and Iterable, giving it the functionality of all three
    @author Zachary Keller
    @version final
*/
public class LinkedList<E> implements Stack<E>, Queue<E>, Iterable<E>
{
    /**
        A pointer to the first ListNode in the LinkedList
    */
    public ListNode<E> head;

    /**
        A pointer to the last ListNode in the LinkedList
    */
    public ListNode<E> tail;

    /**
        The length of the LinkedList, aka the number of ListNodes
    */
    private int size;

    /**
        Default Constructor, initializes head and tail pointers to null
    */
    public LinkedList()
    {
        head = null;
        tail = null;
        size = 0;
    }

    /**
        Constructor that begins a new Linked List with an existing
        Node as the head
        @param h A node that will become the head of the new Linked List
    */
    public LinkedList(ListNode<E> h)
    {
        this();
        add(h.getItem());
    }

    /**
        Copy Constructor, makes a copy of an existing Linked List
        @param list The Linked List to be copied
    */
    public LinkedList(LinkedList<E> list)
    {
        ListNode<E> node = list.head;
        while (node != null)
        {
            add(node.getItem());
            node = node.getNext();
        }
    }

    /**
        Returns the size of the Linked List
        @return The size of the Linked List
    */
    public int size()
    {
        return size;
    }

    /**
        adds a new item to a specific spot in the Linked List
        @param index Where the item should be added
        @param item the thing of type E being added
        @return For a successful addition
    */
    public boolean add( int index, E item)
    {
        // Makes sure the index is within the size of the Linked List
        if (index > size || index < 0 )
        {
            throw new IndexOutOfBoundsException("Index " + index + " is not within the size: " + size
        }
        // If the Linked List is empty
        if (tail == null || index == size)
        {
            return add(item);
        }
        ListNode<E> holder = head;
        int num = 0;
        // Finds the List Node before the spot where the item is to be added
        while (num < index - 1)
        {                                    ✓
            holder = holder.getNext();
            num+=1;
        }
        if (index > 0)
        {
            ListNode<E> l = new ListNode<E>(item, holder.getNext());
            holder.setNext(l);
        }
        else //basically if index is 0
        {
            ListNode<E> l = new ListNode<E>(item, head);
            head = l;
        }
        size+=1;
        return true;
    }

    /**
    Creates and returns an iterator
    @return The Iterator
    */
    public Iterator<E> iterator()
    {
        return new LinkedListIterator<E>(head);
    }

    /**
        Adds an item to the end of the Linked List
        @param item the thing to be added
        @return If the addition was successful
    */
    public boolean add(E item)
    {
        ListNode<E> l = new ListNode<E>(item);
        if (tail == null)
        {
            head = l;            ✓
            tail = l;
            size+=1;
            return true;
        }
        tail.setNext(l);
        tail = l;
        size+=1;
        return true;
    }

    /**
        Removes the ListNode (and therefore item within the ListNode)
        from a given index
        @param index The place that will be removed
        @return The value previously at the index
    */
    public E remove(int index)
    {
        // Makes sure a proper index was used
        if (index > size || index < 0 )
        {
            throw new IndexOutOfBoundsException("Index " + index + " is not within the size: " + size
        }
        ListNode<E> node = head;
        E returner;
        // If you are just trying to remove the head
        if (index == 0)
        {
            removeFirst();
        }
        int num = 0;
        // Gets to the List Node before the one to be removed
        while (num < index - 1)
        {
            node = node.getNext();
            num+=1;
        }
        returner = node.getNext().getItem();
        node.setNext(node.getNext().getNext());
        // Case for if the tail is being removed
        if (index == size - 1)
        {
            tail = node;
        }
        size -= 1;
        return returner;
    }

    /**
        Removes the first instance of a given value
        @param item the desired value to be removed
        @return For a successful removal
    */
    public boolean remove(E item)
    {
        if (! contains(item))
            return false;
        remove(indexOf(item));
        return true;

    }

    /**
        Checks to see if a given value is in the Linked List
        @param object the item that is being checked for
        @return Whether or not the item is contained within the Linked List
    */
    public boolean contains(E object)
    {
        return indexOf(object) != -1;
    }

    /**
        Returns the index of the first instance of an object
        @param object The item that is being checked for
        @return The index of the object if it is in the List, -1 if it is not in the List
    */
    public int indexOf (E object)
    {
        ListNode<E> node = head;
        int num = 0;
        while (num < size)
        {
            if (object == null)   ✓
            {
                 if (node.getItem() == null)
                    return num;
            }
            else
            {
                if (object.equals(node.getItem()))
                    return num;
            }
            node = node.getNext();
            num +=1;
        }
        return -1;
    }

    /**
        Empties the LinkedList
    */
    public void clear()
    {
        head = null;
        tail = null;
        size = 0;
    }

    /**
        Returns the Item at a given index
        @param index The spot to be gotten
        @return The Item at the desired spot
    */
    public E get(int index)
    {
        if (index > size || index < 0 )
        {
            throw new IndexOutOfBoundsException("Index " + index + " is not within the size: " + size
        }
        ListNode<E> node = head;
        int num = 0;
        while (num < index)
        {
            node = node.getNext();
            num+=1;
        }
        return node.getItem();
    }

    /**
    Inserts an item at a given location regardless of what is already there
    @param o the item to be placed
    @param i The spot for the item
    @return The item that was previously in that spot
    */
    public E set(int i , E o)
    {
        add(i, o);
        E holder = get(i+1);
        remove(i + 1);
        return holder;
    }
```

This seems pretty inefficient, even if 3n vs n is essentially the same.

```

    /**
    Identifies whether the Linked List is empty; That is to say its size is 0
    @return Whether or not it is empty
    */
    public boolean isEmpty()
    {
        return (head == null);
    }

    /**
    Returns a string representation of the Linked List
    @return The string representation of the Linked List
    */
    public String toString()
    {
        ListNode<E> node = head;
        String s = "";
        while (node != null)
        {
            s += node.toString();
```

```java
            s+= "\n";
            node = node.getNext();
        }
        return s;

    }

    /**
        Adds and item to the beginning of the linked List- resets the head
        @param item The value to be pushed
    */
    public void push(E item)
    {
        addFirst(item);
    }

    /**
        Removes and returns the head of the Linked List, adjusts accordingly
        @return the value of the head / the first item
    */
    public E pop()
    {
        return removeFirst();
    }

    /**
        Returns what is first in the Linked List, aka the head Node
        BUT does not actually change anything
        @return Head Node
    */
    public E peek()
    {
        return get(0);
    }

    /**
        Adds an item to the end of the Linked List
        @param item The thing being added ("offered")
    */
    public void offer(E item)
    {
        addLast(item);
    }

    /**
        Removes and returns the head of the Linked List, adjusts accordingly
        @return the value of the head / the first item
    */
    public E poll()
    {
        return removeFirst();
    }

    /**
        adds an item to the beginning of the Linked List, adjusts the head accordingly
        @param item the object being added to the List
    */
    public void addFirst(E item)
    {
        add(0, item);
    }

    /**
        Adds an item to the end of the Linked List
        @param item The object being added to the List
    */
    public void addLast(E item)
    {
        add(item);
    }

    /**
        Removes and returns the first element in the Linked List
        @return the item that was removed from the List
    */
    public E removeFirst()
    {
        E returner;
        // Makes sure the Linked List is not empty
        if (head == null)
        {
            throw new NoSuchElementException("Linked List is empty");
        }
        returner = head.getItem();
        head = head.getNext();
        size -= 1;
        if (size == 0)
            tail = null;
        return returner;
    }

    /**
        Removes the Last object in the Linked List, and returns it
        @return The Item being removed
    */
    public E removeLast()
    {
        // Makes sure the Linked List is not empty
        if (head == null)
        {
            throw new NoSuchElementException("Linked List is empty");
        }
        return remove(size - 1);
    }

}
```

Great job. Works for all tests.
Grade: A+