

Data Structures

Implementing Lists, Stacks and Queues

(Chapter 24)

What is a Data Structure?

- A data structure is a collection of data organized in some fashion.
- The structure not only stores data but also supports operations for accessing and manipulating the data.
- In object-oriented thinking, a data structure, also known as a **container or container object**, is an object that stores other objects, referred to as data or elements.

Lists

A list is a popular data structure to store data in sequential order. For example, a list of students, a list of available rooms, a list of cities, and a list of books, etc. can be stored using lists.

The common operations on a list are usually the following:

- **Retrieve** an element from this list.
- **Insert** a new element into this list.
- **Delete** an element from this list.
- **Find how many elements** are in this list.
- **Find if** an element **is in** this list.
- **Find if** this list is **empty**.

Two Ways to Implement Lists

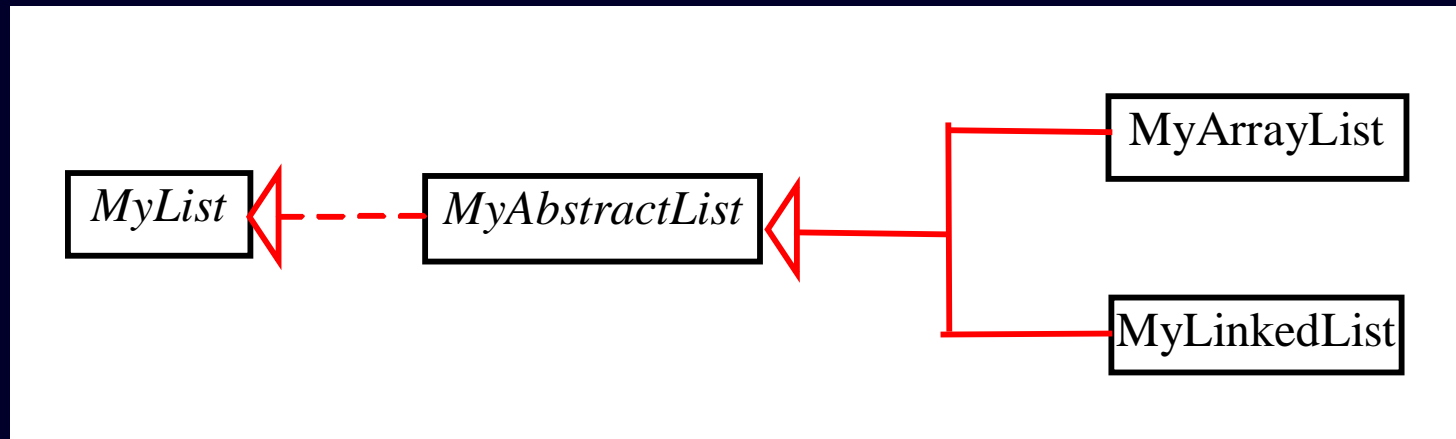
There are two ways to implement a list.

Using arrays. The array is dynamically created. If the capacity of the array is exceeded, create a new larger array and copy all the elements from the current array to the new array.


Using linked list. A linked structure consists of nodes. Each node is dynamically created to hold an element. All the nodes are linked together to form a list.

Design of ArrayList and LinkedList

- **MyArrayList** and **MyLinkedList** have common operations, but different data fields.
- The common operations can be generalized in an interface or an abstract class.



MyList Interface and MyAbstractList Class

<div>«interface» <i>MyList</i><E></div> <div>+add(e: E) : void +add(index: int, e: E) : void +clear(): void +contains(e: E): boolean +get(index: int) : E +indexOf(e: E) : int +isEmpty(): boolean +lastIndexOf(e: E) : int +remove(e: E): boolean +size(): int +remove(index: int) : E +set(index: int, e: E) : E</div>	<p>Appends a new element at the end of this list.</p> <p>Adds a new element at the specified index in this list.</p> <p>Removes all the elements from this list.</p> <p>Returns true if this list contains the element.</p> <p>Returns the element from this list at the specified index.</p> <p>Returns the index of the first matching element in this list.</p> <p>Returns true if this list contains no elements.</p> <p>Returns the index of the last matching element in this list.</p> <p>Removes the element from this list.</p> <p>Returns the number of elements in this list.</p> <p>Removes the element at the specified index and returns the removed element.</p> <p>Sets the element at the specified index and returns the element you are replacing.</p>
<div> <i>MyAbstractList</i><E></div> <div>#size: int #MyAbstractList() #MyAbstractList(objects: E[]) +add(e: E) : void +isEmpty(): boolean +size(): int +remove(e: E): boolean</div>	<p>The size of the list.</p> <p>Creates a default list.</p> <p>Creates a list from an array of objects.</p> <p>Implements the add method.</p> <p>Implements the isEmpty method.</p> <p>Implements the size method.</p> <p>Implements the remove method.</p>

```

public interface MyList<E> extends java.lang.Iterable<E> {
    /** Add a new element at the end of this list */
    public void add(E e);

    /** Add a new element at the specified index in this list */
    public void add(int index, E e);

    /** Clear the list */
    public void clear();

    /** Return true if this list contains the element */
    public boolean contains(E e);

    /** Return the element from this list at the specified index */
    public E get(int index);

    /** Return the index of the first matching element in this list.
     * Return -1 if no match. */
    public int indexOf(E e);

    /** Return true if this list contains no elements */
    public boolean isEmpty();

    /** Return the index of the last matching element in this list
     * Return -1 if no match. */
    public int lastIndexOf(E e);

    /** Remove the first occurrence of the element o from this list.
     * Shift any subsequent elements to the left.
     * Return true if the element is removed. */
    public boolean remove(E e);

    /** Remove the element at the specified position in this list
     * Shift any subsequent elements to the left.
     * Return the element that was removed from the list. */
    public E remove(int index);

    /** Replace the element at the specified position in this list
     * with the specified element and returns the old element. */
    public Object set(int index, E e);

    /** Return the number of elements in this list */
    public int size();
}

```

Code for MyList Interface

```

public abstract class MyAbstractList<E> implements MyList<E> {
    protected int size = 0; // The size of the list

    /** Create a default list */
    protected MyAbstractList() {
    }

    /** Create a list from an array of objects */
    protected MyAbstractList(E[] objects) {
        for (int i = 0; i < objects.length; i++)
            add(objects[i]);
    }

    @Override /** Add a new element at the end of this list */
    public void add(E e) {
        add(size, e);
    }

    @Override /** Return true if this list contains no elements */
    public boolean isEmpty() {
        return size == 0;
    }

    @Override /** Return the number of elements in this list */
    public int size() {
        return size;
    }

    @Override /** Remove the first occurrence of the element e
     * from this list. Shift any subsequent elements to the left.
     * Return true if the element is removed. */
    public boolean remove(E e) {
        if (indexOf(e) >= 0) {
            remove(indexOf(e));
            return true;
        }
        else
            return false;
    }
}

```

Code for MyAbstractList Class

Array Lists

Array lists are implemented using arrays. An array is a fixed-size data structure. Once an array is created, its size cannot be changed. Nevertheless, you can still use array to implement dynamic data structures.

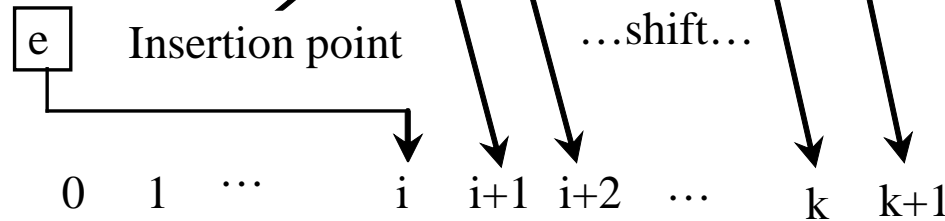
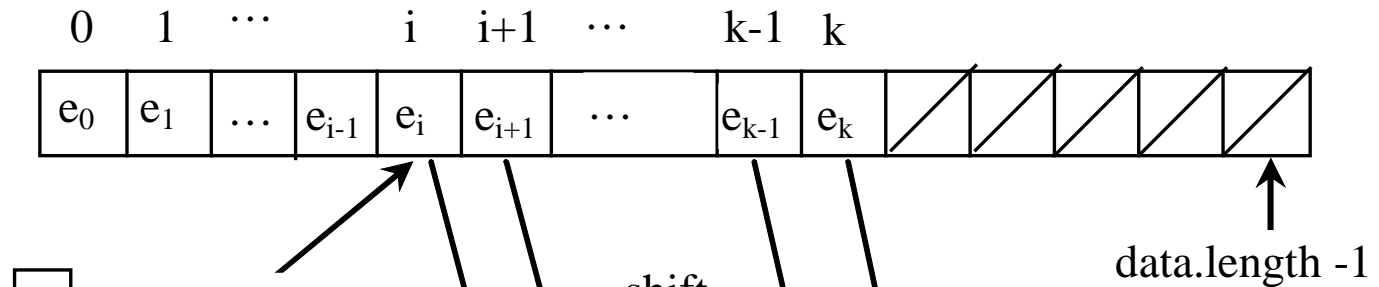
The trick is to create a new larger array to replace the current array if the current array cannot hold new elements in the list.

Initially, an array, say `data` of `Object[]` type, is created with a default size. When inserting a new element into the array, first ensure there is enough room in the array. If not, create a new array with the size as twice as the current one. Copy the elements from the current array to the new array. The new array now becomes the current array.

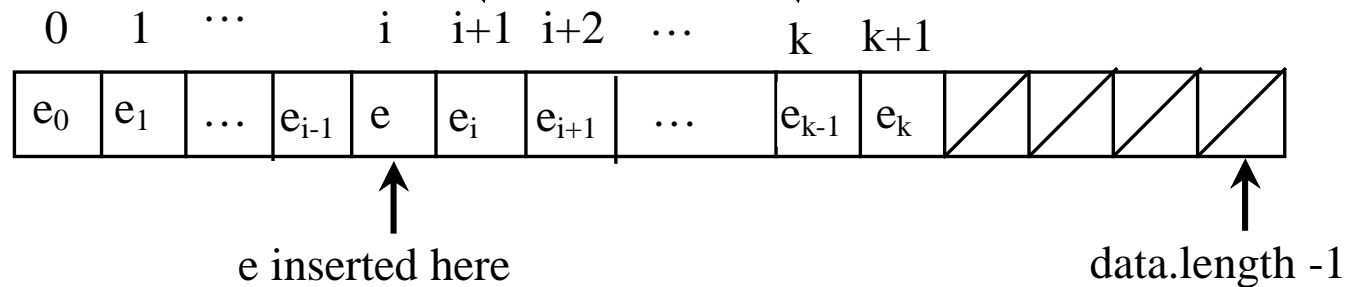
Insertion

Before inserting a new element at a specified index, shift all the elements after the index to the right and increase the list size by 1.

Before inserting
e at insertion point i



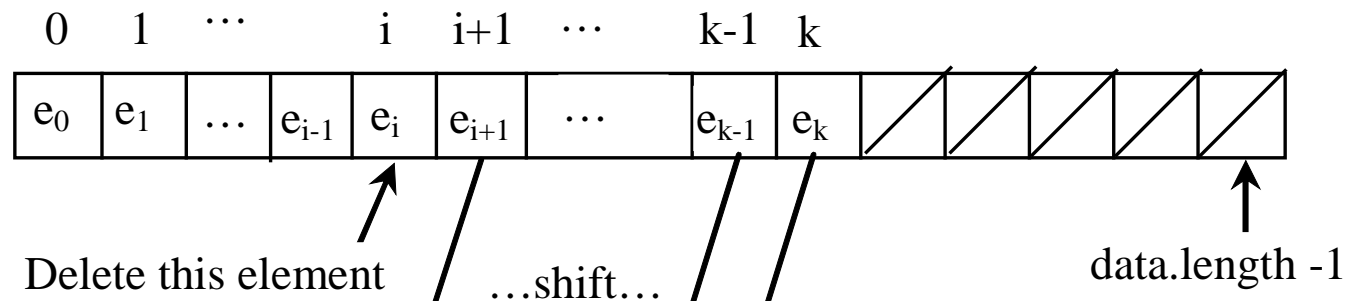
After inserting
e at insertion point i,
list size is
incremented by 1



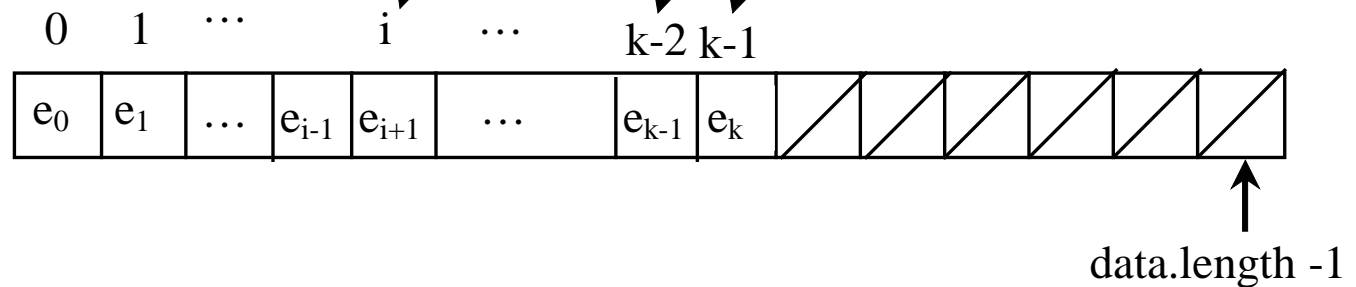
Deletion

To remove an element at a specified index, shift all the elements after the index to the left by one position and decrease the list size by 1.

Before deleting the element at index i

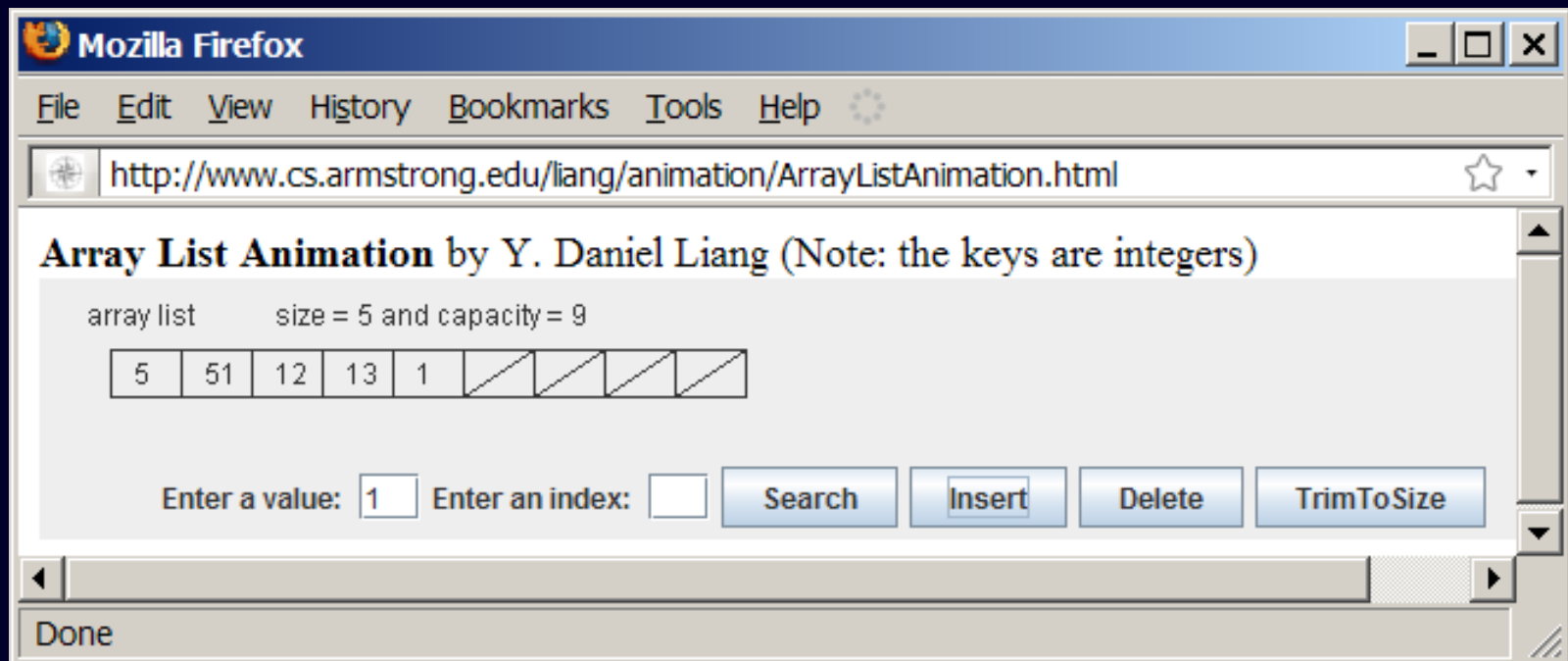


After deleting the element, list size is decremented by 1

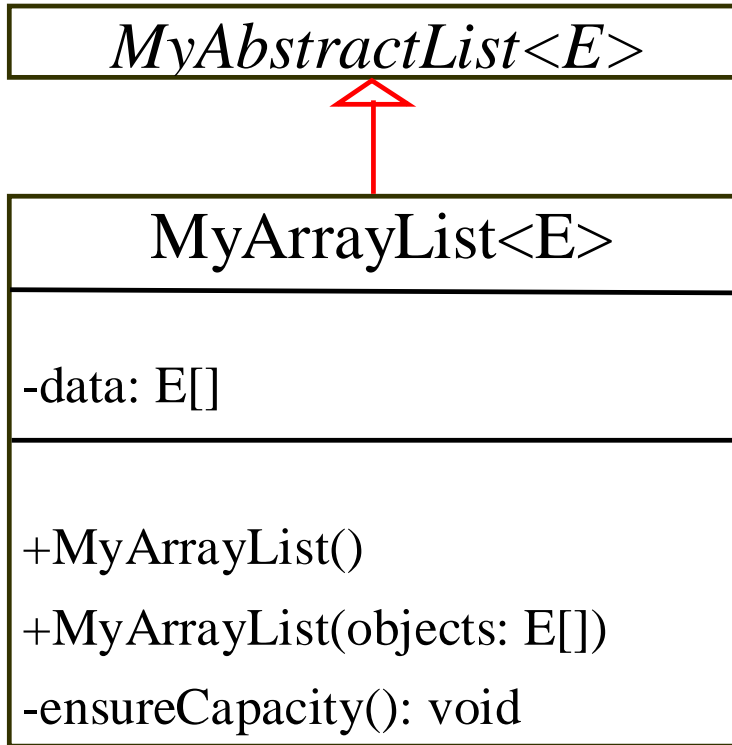


Array List Animation

www.cs.armstrong.edu/liang/animation/



Implementing MyArrayList



Creates a default array list.

Creates an array list from an array of objects.

Doubles the current array size if needed.

```

public class MyArrayList<E> extends MyAbstractList<E> {
    public static final int INITIAL_CAPACITY = 16;
    private E[] data = (E[])new Object[INITIAL_CAPACITY];

    /** Create a default list */
    public MyArrayList() {
    }

    /** Create a list from an array of objects */
    public MyArrayList(E[] objects) {
        for (int i = 0; i < objects.length; i++)
            add(objects[i]);
    }

    @Override /** Add a new element at the specified index */
    public void add(int index, E e) {
        checkIndex(index);

        ensureCapacity();

        // Move the elements to the right after the specified index
        for (int i = size - 1; i >= index; i--)
            data[i + 1] = data[i];

        // Insert new element to data[index]
        data[index] = e;

        // Increase size by 1
        size++;
    }
}

```

Code for MyArrayList Class

```

/** Create a new larger array, double the current size + 1 */
private void ensureCapacity() {
    if (size >= data.length) {
        E[] newData = (E[]) (new Object[size * 2 + 1]);
        System.arraycopy(data, 0, newData, 0, size);
        data = newData;
    }
}

@Override /** Clear the list */
public void clear() {
    data = (E[]) new Object[INITIAL_CAPACITY];
    size = 0;
}

@Override /** Return true if this list contains the element */
public boolean contains(E e) {
    for (int i = 0; i < size; i++)
        if (e.equals(data[i])) return true;

    return false;
}

@Override /** Return the element at the specified index */
public E get(int index) {
    checkIndex(index);
    return data[index];
}

private void checkIndex(int index) {
    if (index < 0 || index >= size)
        throw new IndexOutOfBoundsException
            ("Index: " + index + ", Size: " + size);
}

```

Code for
MyArrayList Class cont.

```

@Override /** Return the index of the first matching element
 * in this list. Return -1 if no match. */
public int indexOf(E e) {
    for (int i = 0; i < size; i++)
        if (e.equals(data[i])) return i;

    return -1;
}

@Override /** Return the index of the last matching element
 * in this list. Return -1 if no match. */
public int lastIndexOf(E e) {
    for (int i = size - 1; i >= 0; i--)
        if (e.equals(data[i])) return i;

    return -1;
}

@Override /** Remove the element at the specified position
 * in this list. Shift any subsequent elements to the left.
 * Return the element that was removed from the list. */
public E remove(int index) {
    checkIndex(index);

    E e = data[index];

    // Shift data to the left
    for (int j = index; j < size - 1; j++)
        data[j] = data[j + 1];

    data[size - 1] = null; // This element is now null

    // Decrement size
    size--;

    return e;
}

```

Code for
MyArrayList Class cont.


```

@Override /** Replace the element at the specified position
 * in this list with the specified element. */
public E set(int index, E e) {
    checkIndex(index);
    E old = data[index];
    data[index] = e;
    return old;
}

@Override
public String toString() {
    StringBuilder result = new StringBuilder("[");

    for (int i = 0; i < size; i++) {
        result.append(data[i]);
        if (i < size - 1) result.append(", ");
    }

    return result.toString() + "]";
}

/** Trims the capacity to current size */
public void trimToSize() {
    if (size != data.length) {
        E[] newData = (E[]) (new Object[size]);
        System.arraycopy(data, 0, newData, 0, size);
        data = newData;
    } // If size == capacity, no need to trim
}

```

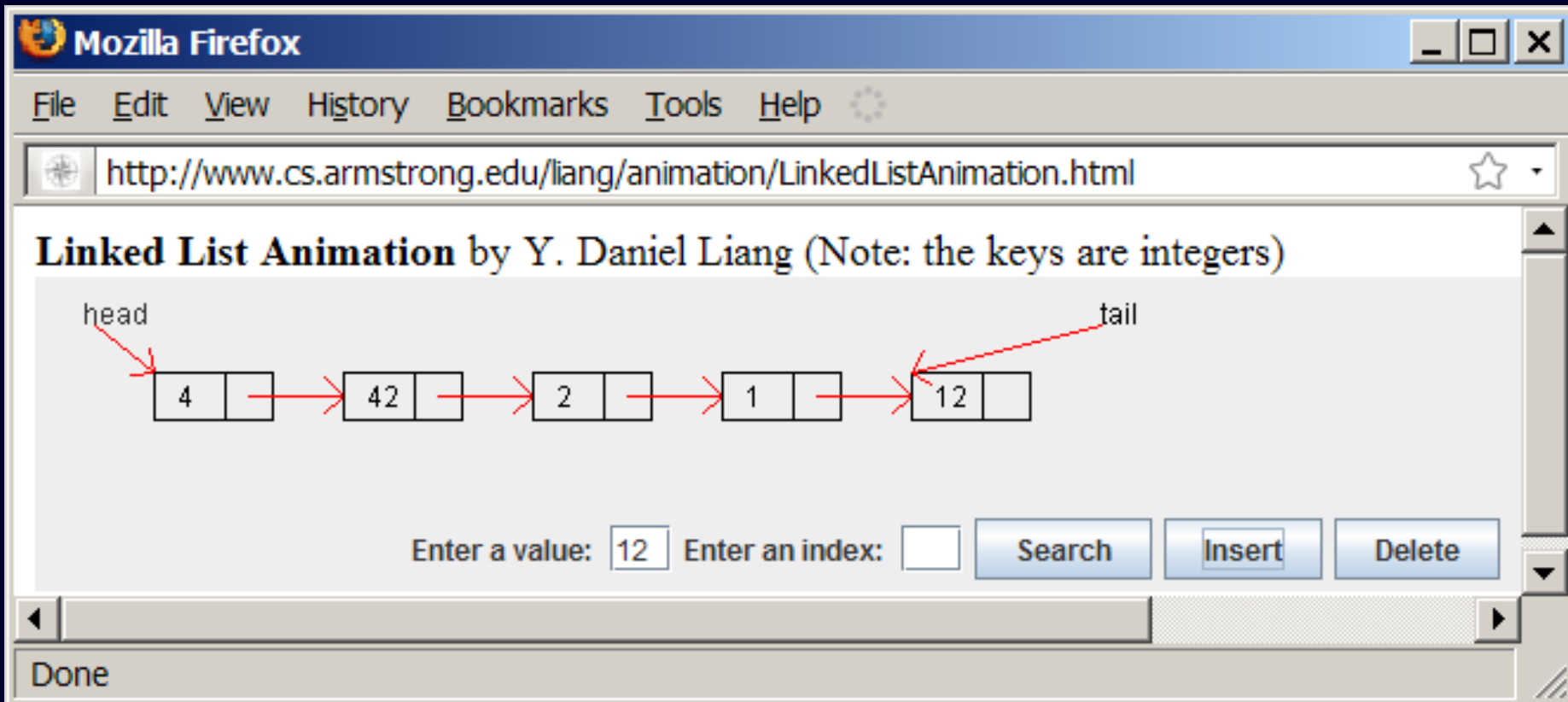
Code for MyArrayList Class cont.

Linked Lists

- Since **MyArrayList** is implemented using an array, the methods **get(int index)** and **set(int index, Object o)** for accessing and modifying an element through an index and the **add(Object o)** for adding an element at the end of the list are efficient.
- However, the methods **add(int index, Object o)** and **remove(int index)** are inefficient because it requires shifting potentially a large number of elements.
- You can use a linked list structure to implement a list to improve efficiency for adding and removing an element anywhere in a list.

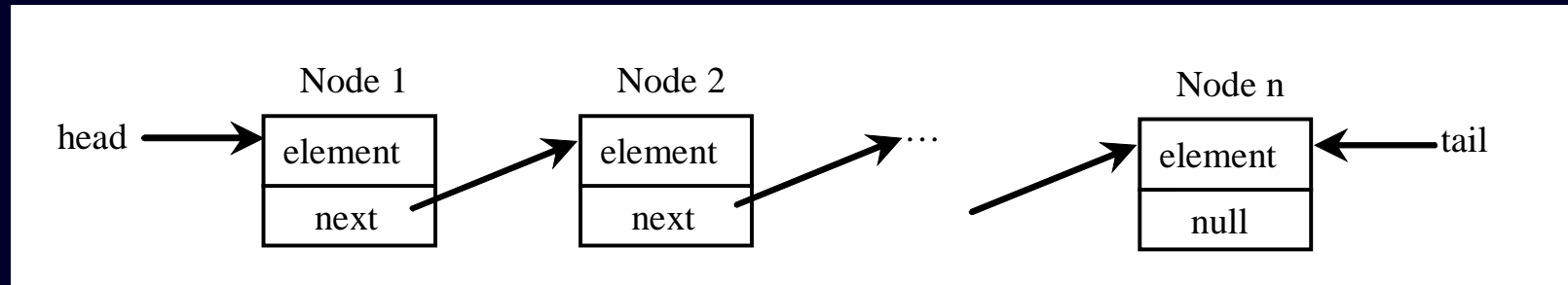
Linked List Animation

www.cs.armstrong.edu/liang/animation/



Nodes in Linked Lists

A linked list consists of nodes. Each node contains an element, and each node is linked to its next neighbor. Thus a node can be defined as a class, as follows:



```
class Node<E> {  
    E element;  
    Node next;  
  
    public Node(E o) {  
        element = o;  
    }  
}
```

Adding Three Nodes

The variable **head** refers to the first node in the list, and the variable **tail** refers to the last node in the list. If the list is empty, both are **null**. For example, you can create three nodes to store three strings in a list, as follows:

Step 1: Declare **head** and **tail**:

```
Node<String> head = null;  
Node<String> tail = null;
```

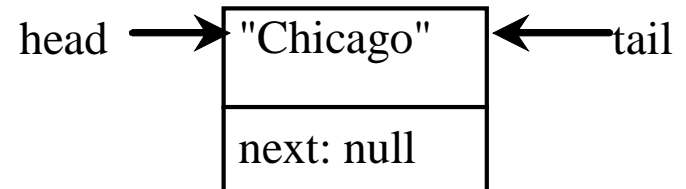
The list is empty now

Adding Three Nodes, cont.

Step 2: Create the first node and insert it to the list:

```
head = new Node<String>( "Chicago" );  
tail = head;
```

After the first node is inserted

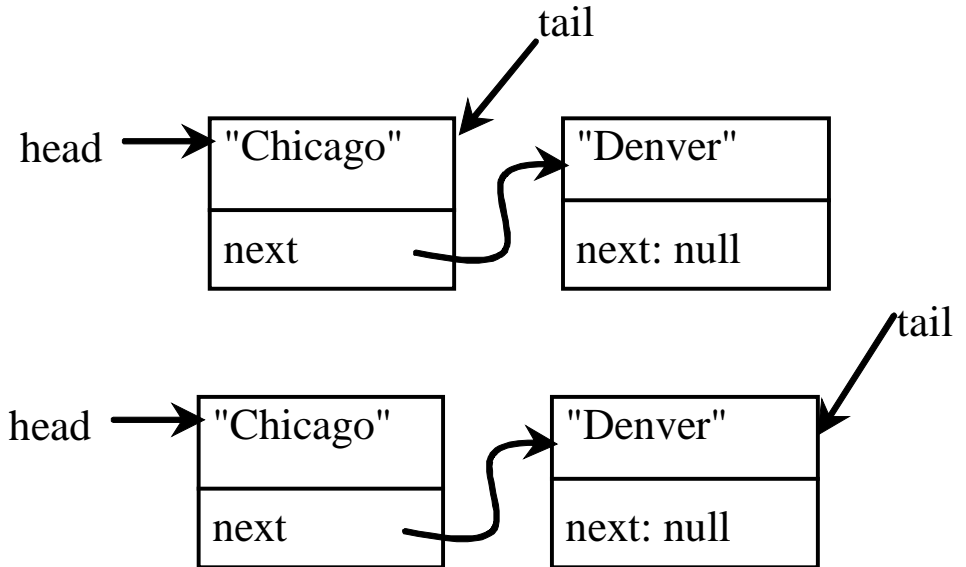


Adding Three Nodes, cont.

Step 3: Create the second node and insert it to the list:

```
tail.next = new Node<String>( "Denver " );
```

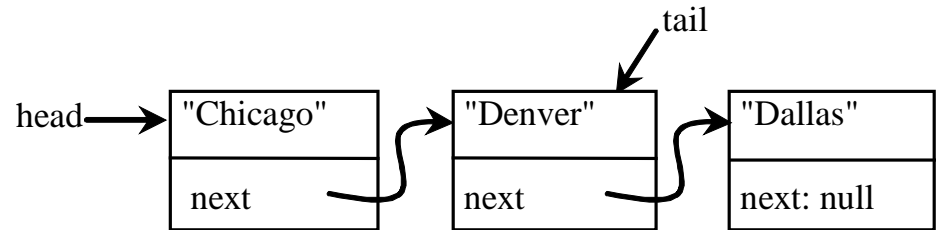
```
tail = tail.next;
```



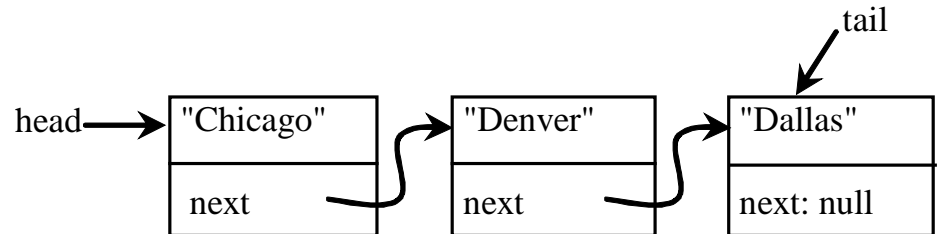
Adding Three Nodes, cont.

Step 4: Create the third node and insert it to the list:

```
tail.next =  
  new Node<String>( "Dallas" );
```



```
tail = tail.next;
```

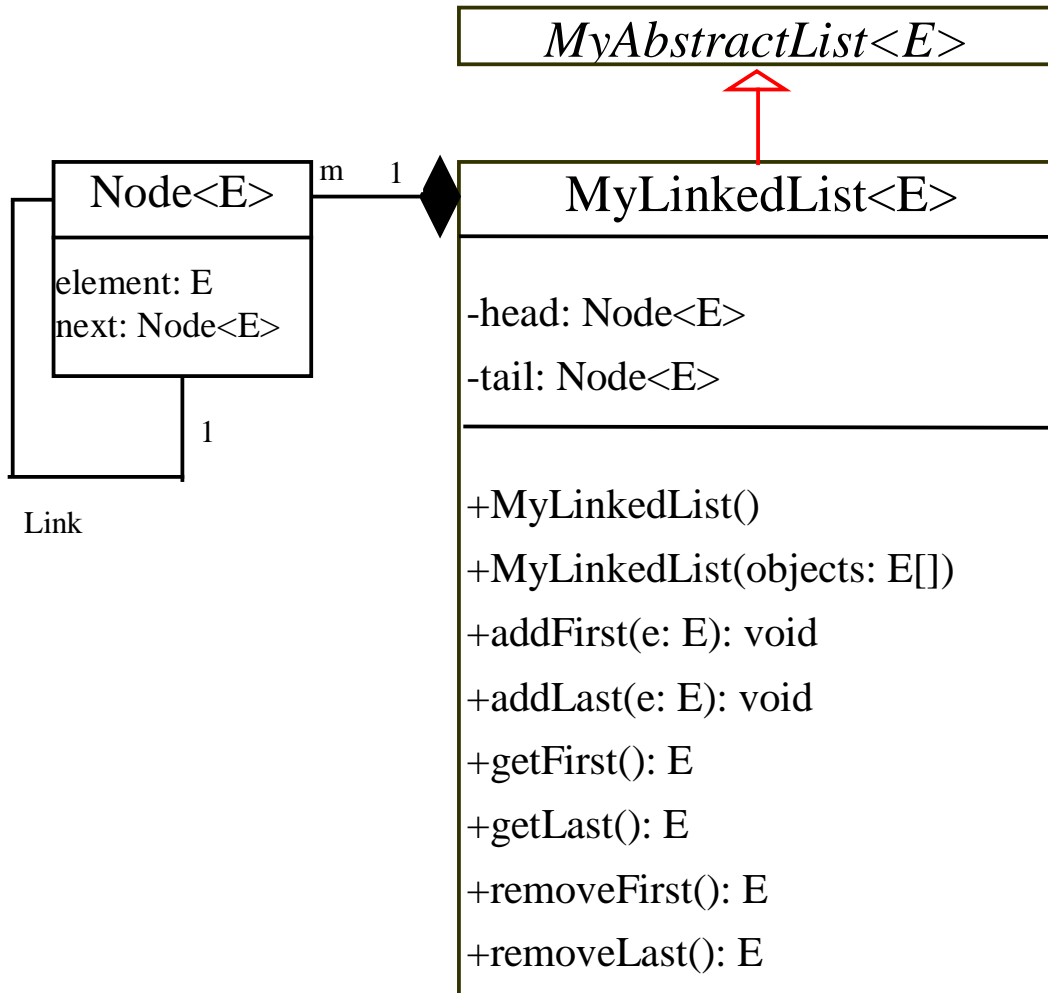


Traversing All Elements in the List

Each node contains the element and a data field named **next** that points to the next element. If the node is the last in the list, its pointer data field **next** contains the value **null**. You can use this property to detect the last node. For example, you may write the following loop to traverse all the nodes in the list.

```
Node<E> current = head;  
while (current != null) {  
    System.out.println(current.element);  
    current = current.next;  
}
```

MyLinkedList



Creates a default linked list.

Creates a linked list from an array of objects.

Adds the object to the head of the list.

Adds the object to the tail of the list.

Returns the first object in the list.

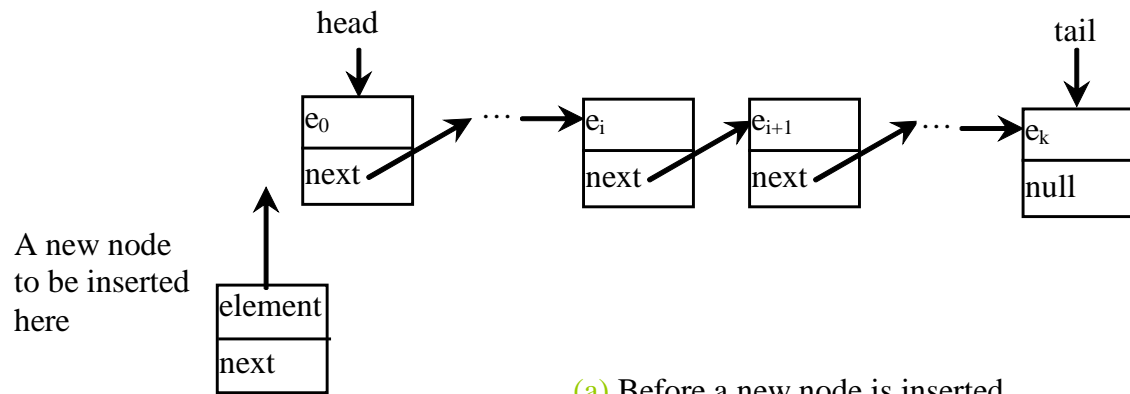
Returns the last object in the list.

Removes the first object from the list.

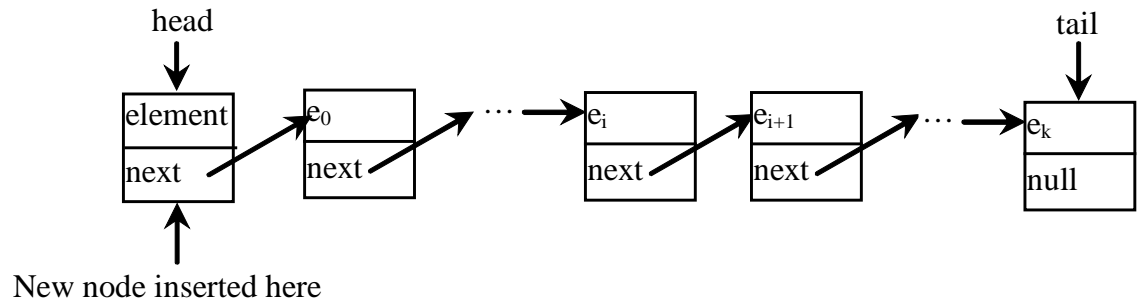
Removes the last object from the list.

Implementing addFirst(E o)

```
public void addFirst(E o) {  
    Node<E> newNode = new Node<E>(o);  
    newNode.next = head;  
    head = newNode;  
    size++;  
    if (tail == null)  
        tail = head;  
}
```



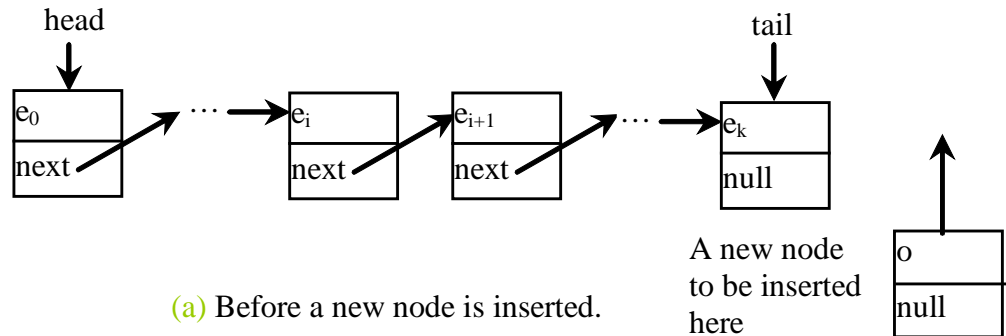
(a) Before a new node is inserted.



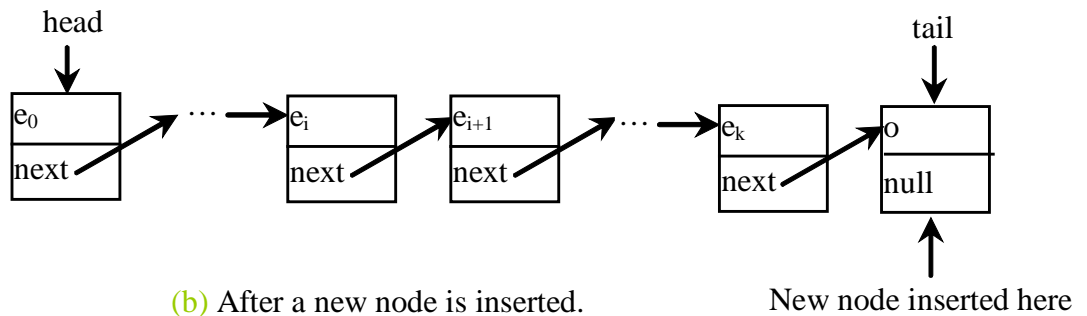
(b) After a new node is inserted.

Implementing addLast(E o)

```
public void addLast(E o) {  
    if (tail == null) {  
        head = tail = new Node<E>(o);  
    }  
    else {  
        tail.next = new Node(o);  
        tail = tail.next;  
    }  
    size++;  
}
```



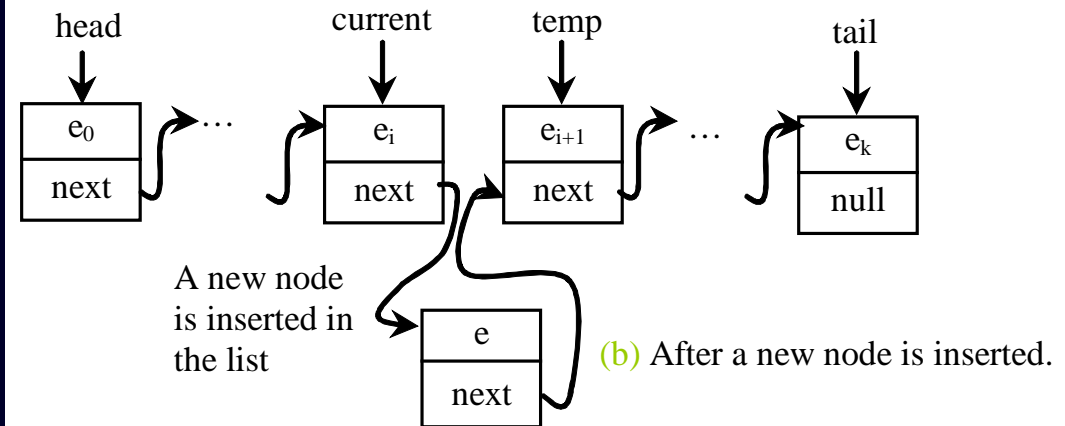
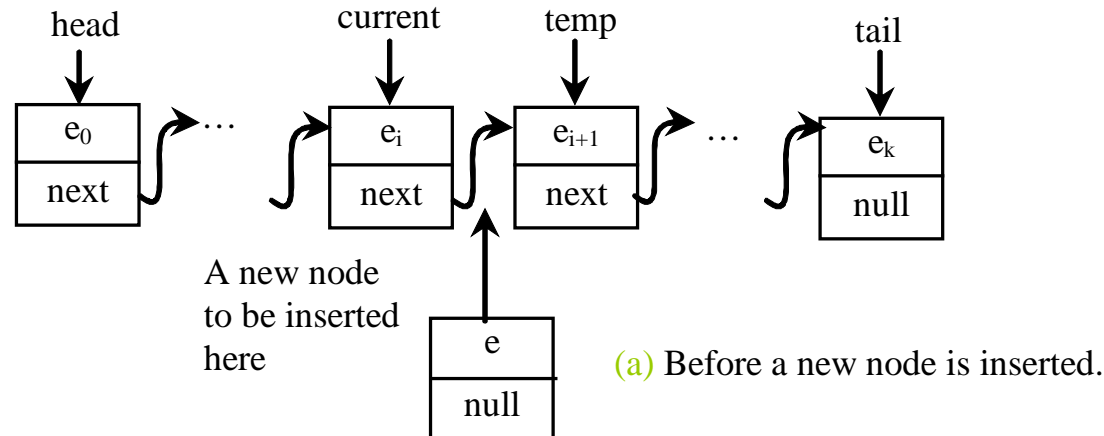
(a) Before a new node is inserted.



(b) After a new node is inserted.

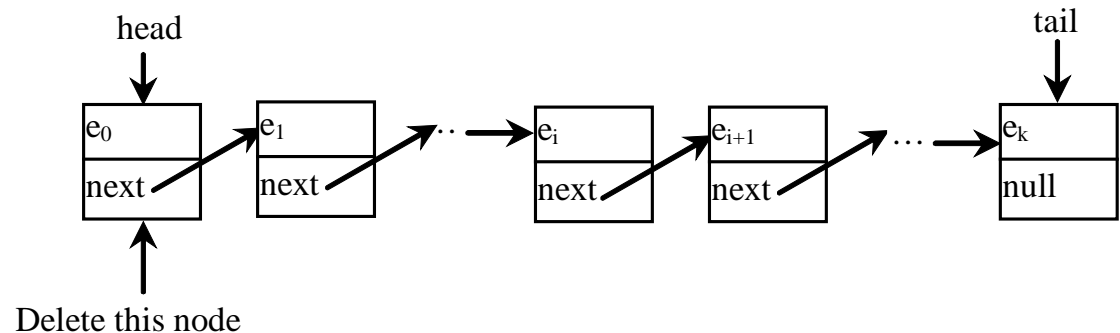
Implementing add(int index, E o)

```
public void add(int index, E o) {  
    if (index == 0) addFirst(o);  
    else if (index >= size) addLast(o);  
    else {  
        Node<E> current = head;  
        for (int i = 1; i < index; i++)  
            current = current.next;  
        Node<E> temp = current.next;  
        current.next = new Node<E>(o);  
        (current.next).next = temp;  
        size++;  
    }  
}
```

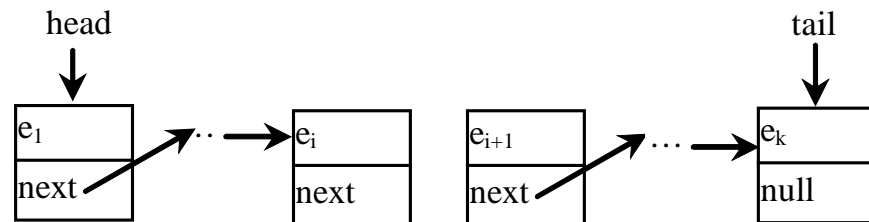


Implementing removeFirst()

```
public E removeFirst() {  
    if (size == 0) return null;  
    else {  
        Node<E> temp = head;  
        head = head.next;  
        size--;  
        if (head == null) tail = null;  
        return temp.element;  
    }  
}
```



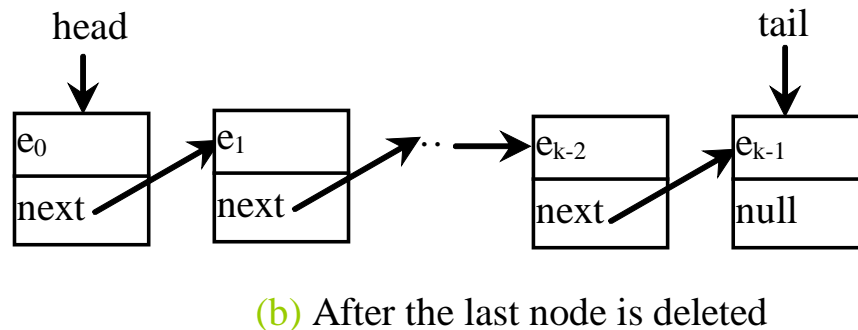
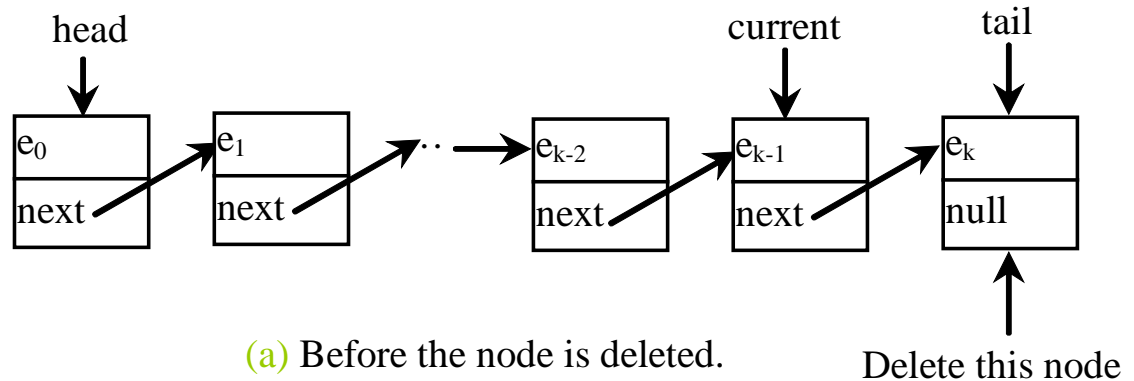
(a) Before the node is deleted.



(b) After the first node is deleted

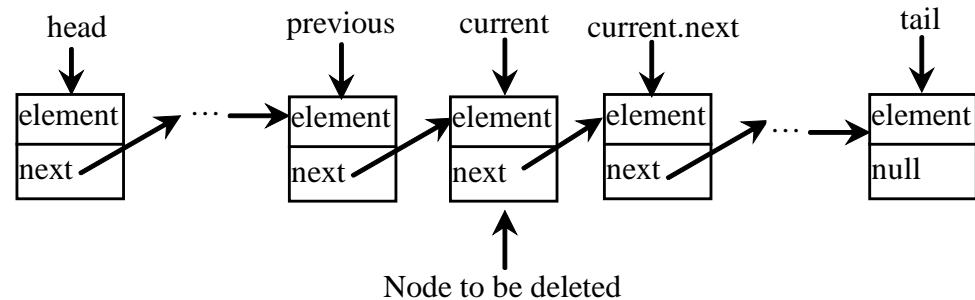
Implementing removeLast()

```
public E removeLast() {  
    if (size == 0) return null;  
    else if (size == 1)  
    {  
        Node<E> temp = head;  
        head = tail = null;  
        size = 0;  
        return temp.element;  
    }  
    else  
    {  
        Node<E> current = head;  
        for (int i = 0; i < size - 2; i++)  
            current = current.next;  
        Node<E> temp = tail;  
        tail = current;  
        tail.next = null;  
        size--;  
        return temp.element;  
    }  
}
```

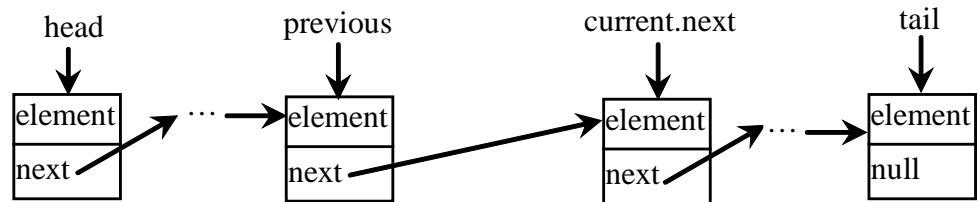


Implementing remove(int index)

```
public E remove(int index) {  
    if (index < 0 || index >= size) return null;  
    else if (index == 0) return removeFirst();  
    else if (index == size - 1) return removeLast();  
    else {  
        Node<E> previous = head;  
        for (int i = 1; i < index; i++) {  
            previous = previous.next;  
        }  
        Node<E> current = previous.next;  
        previous.next = current.next;  
        size--;  
        return current.element;  
    }  
}
```



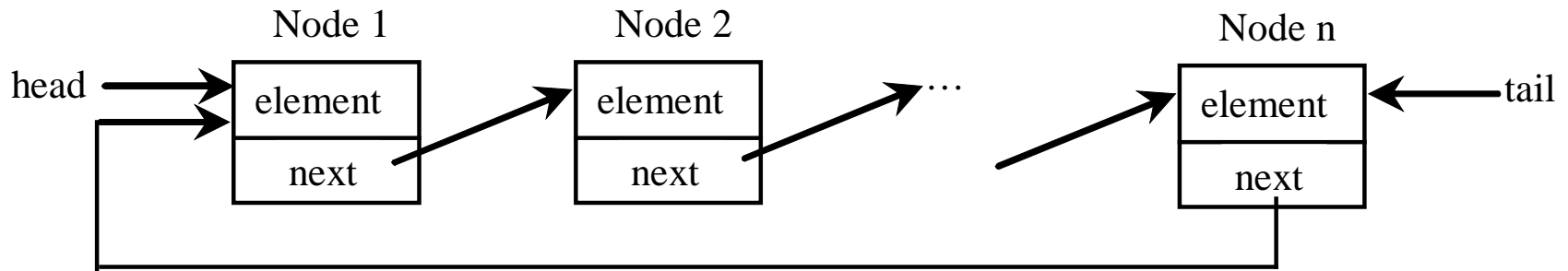
(a) Before the node is deleted.



(b) After the node is deleted.

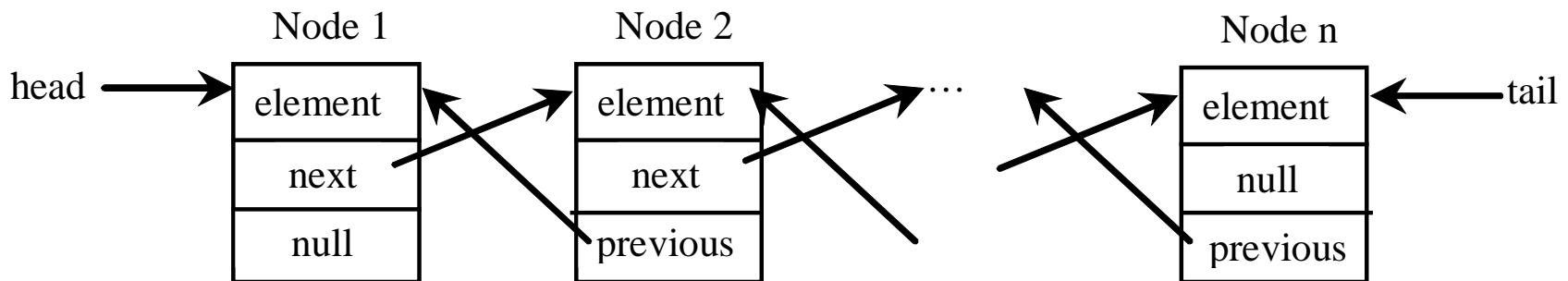
Circular Linked Lists

- A *circular, singly linked list* is like a singly linked list, except that the pointer of the last node points back to the first node.



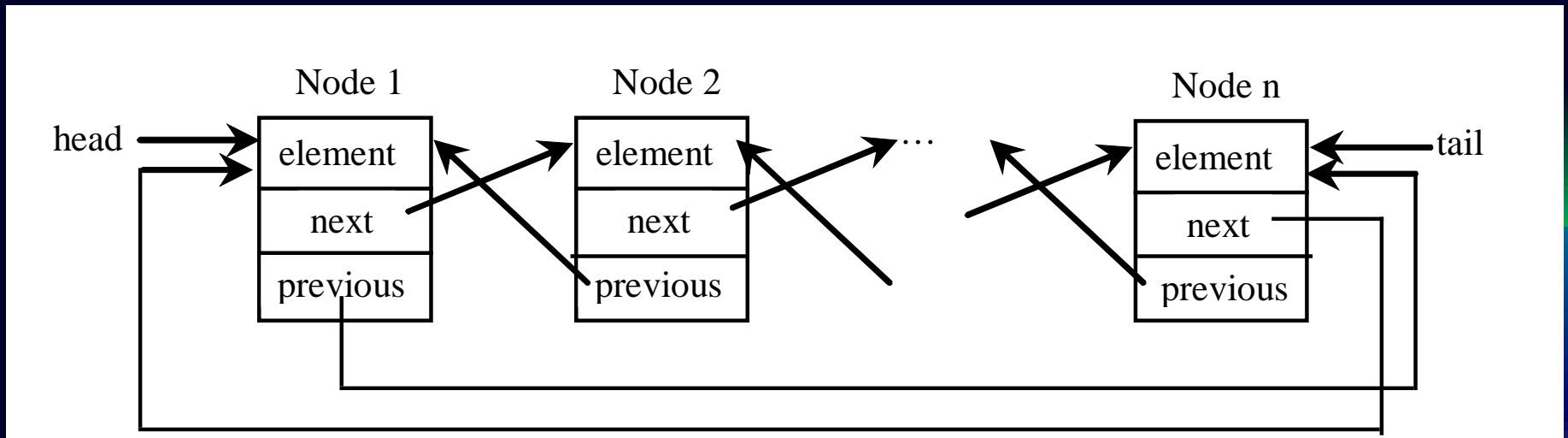
Doubly Linked Lists

- A *doubly linked list* contains the nodes with two pointers. One points to the next node and the other points to the previous node. These two pointers are conveniently called *a forward pointer* and *a backward pointer*. So, a doubly linked list can be traversed forward and backward.



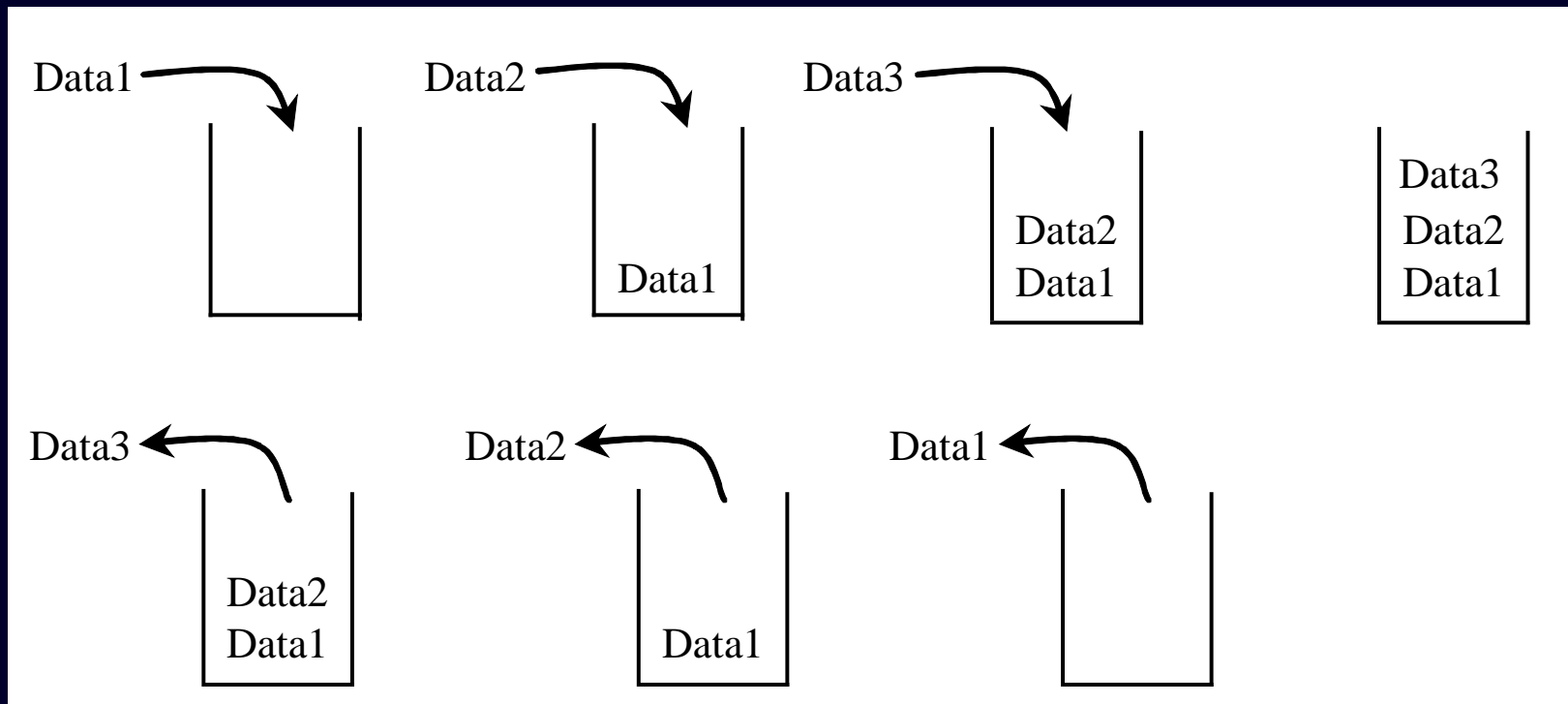
Circular Doubly Linked Lists

- A *circular, doubly linked list* is doubly linked list, except that the forward pointer of the last node points to the first node and the backward pointer of the first pointer points to the last node.



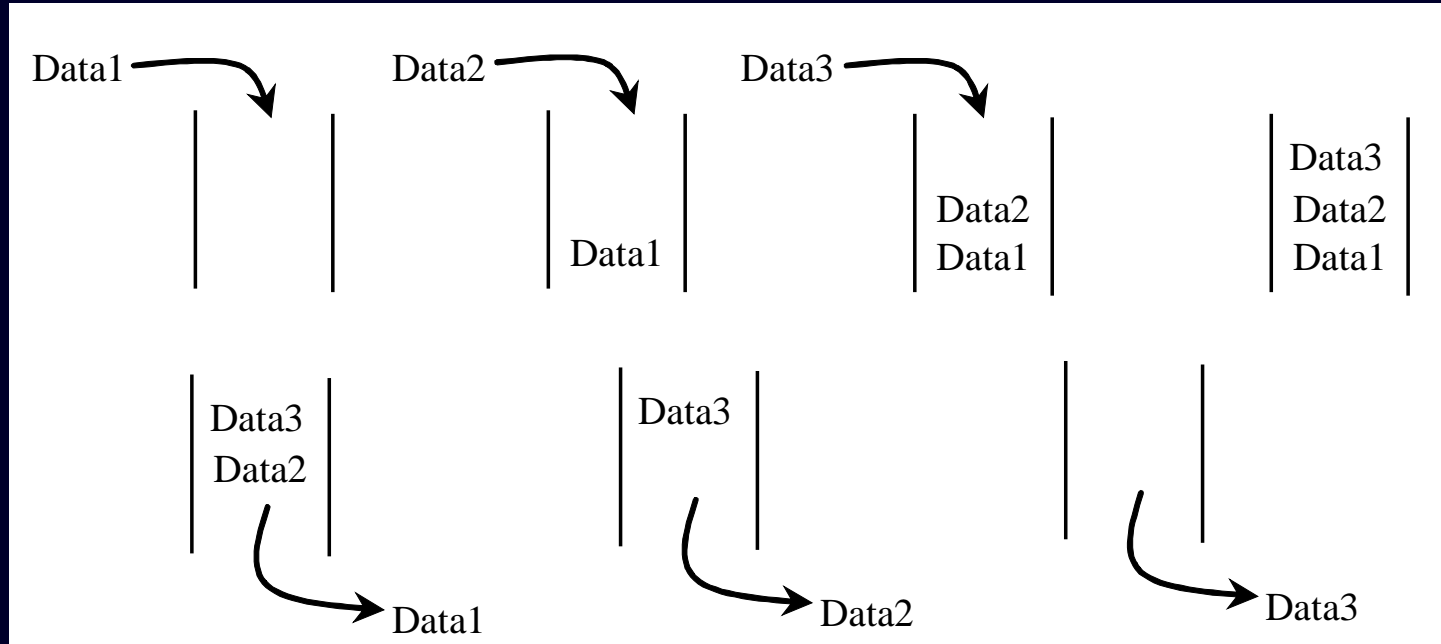
Stacks

A stack can be viewed as a special type of list, where the elements are accessed, inserted, and deleted only from the end, called the top, of the stack.



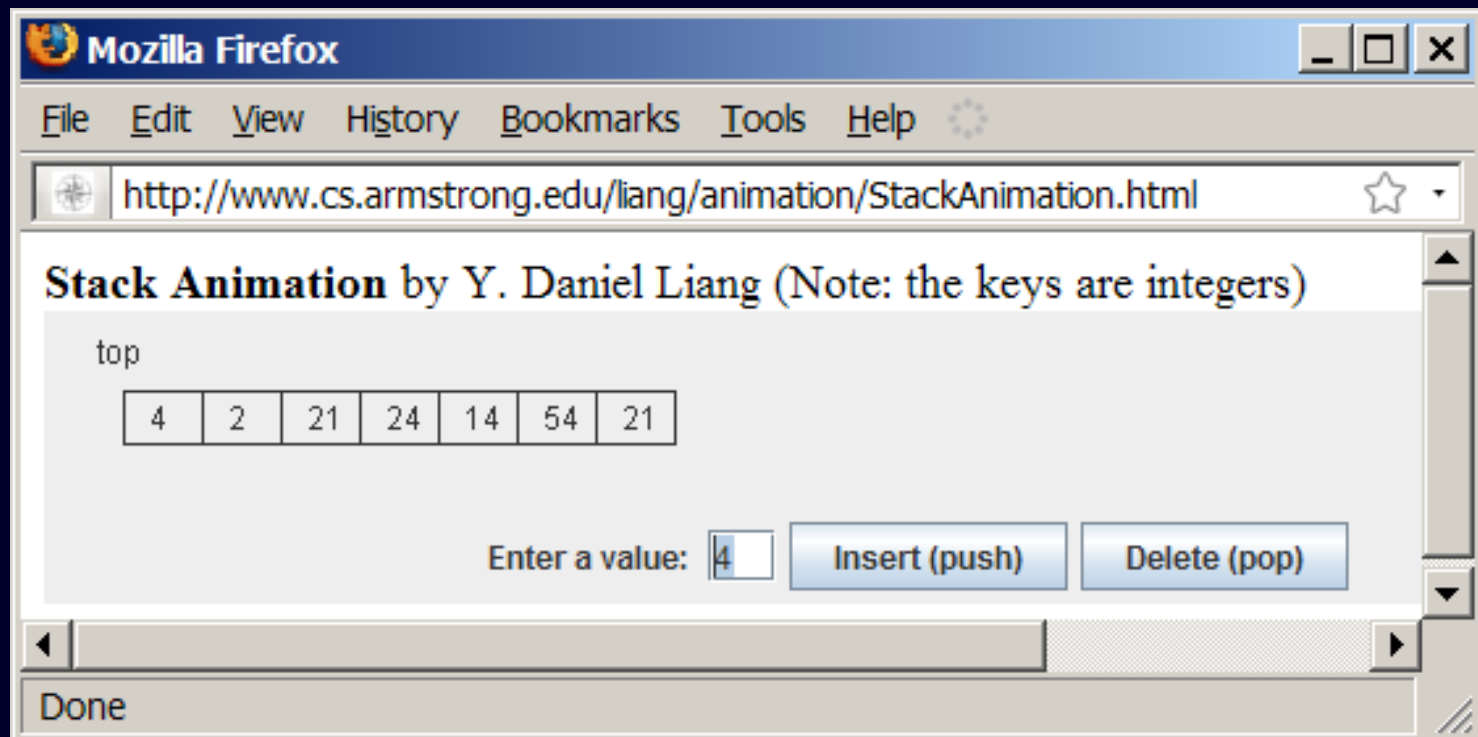
Queues

A queue represents a waiting list. A queue can be viewed as a special type of list, where the elements are inserted into the end (tail) of the queue, and are accessed and deleted from the beginning (head) of the queue.



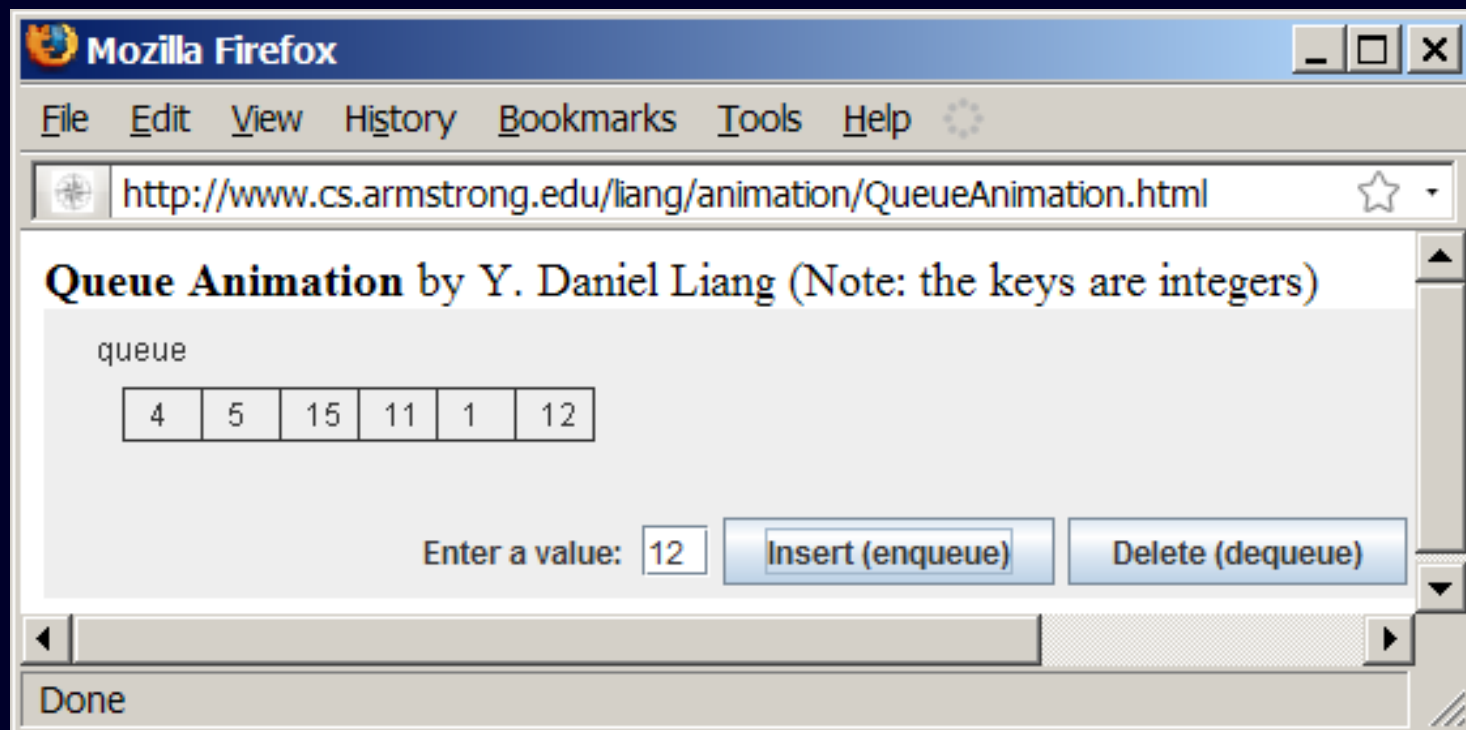
Stack Animation

www.cs.armstrong.edu/liang/animation/



Queue Animation

www.cs.armstrong.edu/liang/animation/



Implementing Stacks and Queues

- Use an array list to implement Stack
 - Since the insertion and deletion operations on a stack are made only at the end of the stack, using an array list to implement a stack is more efficient than a linked list.
- Use a linked list to implement Queue
 - Since deletions are made at the beginning of the list, it is more efficient to implement a queue using a linked list than an array list.

Design of the Stack and Queue Classes

There are two ways to design the stack and queue classes:

- Using inheritance: You can define the stack class by extending the array list class, and the queue class by extending the linked list class.



- Using composition: You can define an array list as a data field in the stack class, and a linked list as a data field in the queue class.



Composition is Better

Both designs are fine, but using composition is better because it enables you to define a complete new stack class and queue class without inheriting the unnecessary and inappropriate methods from the array list and linked list.

MyStack and MyQueue

GenericStack<E>

-list: java.util.ArrayList<E>

An array list to store elements.

+GenericStack()

Creates an empty stack.

+getSize(): int

Returns the number of elements in this stack.

+peek(): E

Returns the top element in this stack.

+pop(): E

Returns and removes the top element in this stack.

+push(o: E): void

Adds a new element to the top of this stack.

+isEmpty(): boolean

Returns true if the stack is empty.

GenericQueue<E>

-list: MyLinkedList<E>

+enqueue(e: E): void

Adds an element to this queue.

+dequeue(): E

Removes an element from this queue.

+getSize(): int

Returns the number of elements from this queue.

Programming Challenge

Implement MyStack and MyQueue classes as defined in the previous slide.

Programming Challenge

The **MyLinkedList** class in your book is a one-way directional linked list that enables one-way traversal of the list. Modify the **Node** class to add the new field `previous` to refer to the previous node in the list, as follows:

```
public class Node<E> {  
    E element;  
    Node<E> next;  
    Node<E> previous;  
  
    public Node(E e) {  
        element = e;  
    }  
}
```

Implement a new class named **MyTwoWayLinkedList** that uses a doubly linked list to store elements.