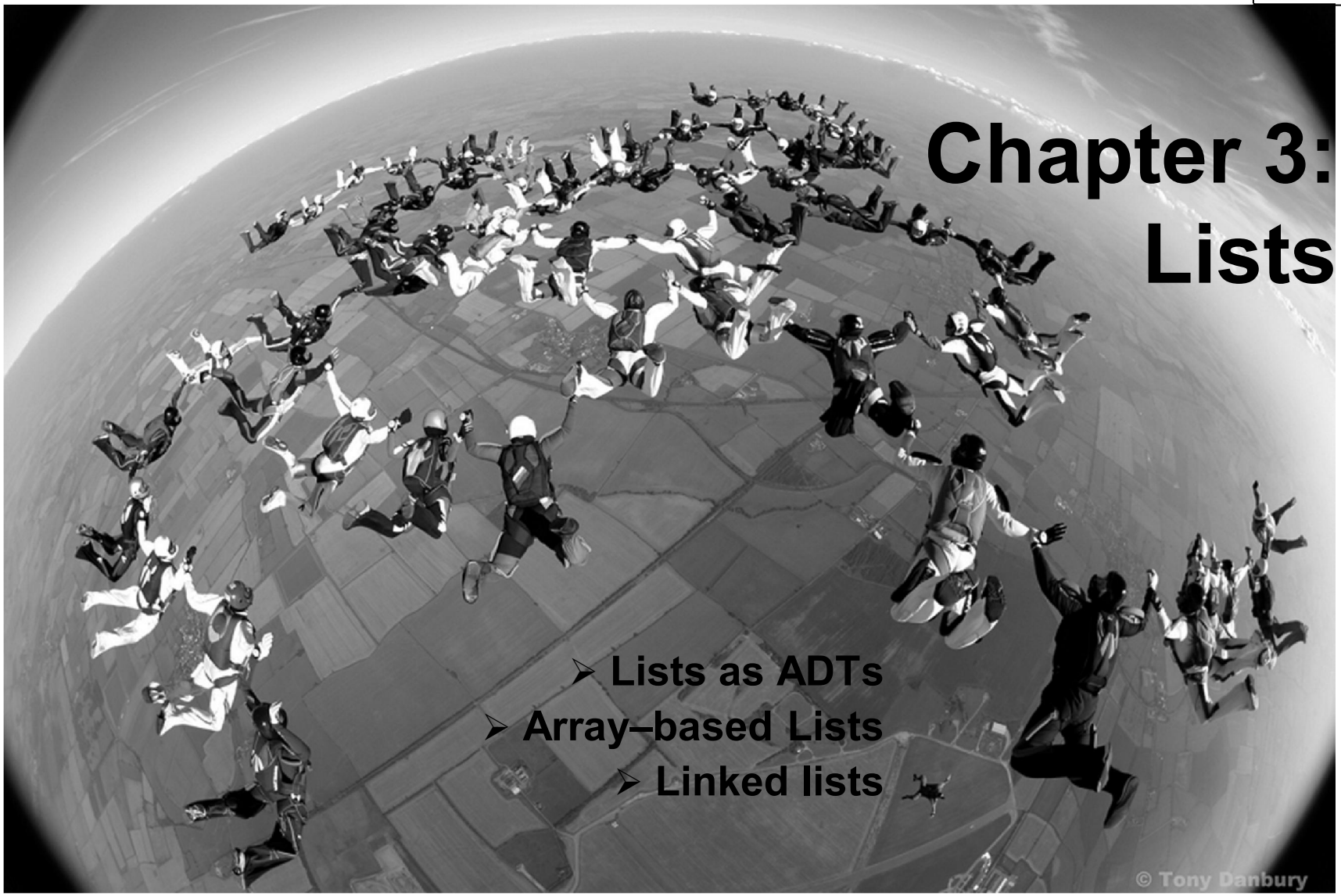
Abu Dhabi University
College of Engineering and Computer Science

جامعة أبوظبي
ABU DHABI UNIVERSITY



Chapter 3: Lists

- Lists as ADTs
- Array–based Lists
- Linked lists

ArrayList Implementation (3)

- ***MyList*** **interface**

```
public interface MyList<E> extends java.lang.Iterable<E> {

public void add(E e);            /** Add a new element at the end of this list */
public void add(int index, E e); /** Add a new element at the specified index in this list */
public void clear();             /** Clear the list */
public boolean contains(E e);    /** Return true if this list contains the element */
public E get(int index);         /** Return the element from this list at the specified index */
public int indexOf(E e);         /** Return the index of the first matching element in this list.
                                   *  Return -1 if no match. */
public boolean isEmpty();        /** Return true if this list doesn't contain any elements */
public int lastIndexOf(E e);     /** Return the index of the last matching element in this list
                                   *  Return -1 if no match. */
public boolean remove(E e);      /** 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 E remove(int index);      /** 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 Object set(int index, E e); /** Replace the element at the specified position in this list
                                   *  with the specified element and return the old element. */
public int size();               /** Return the number of elements in this list */

  }
```

ArrayList Implementation (3)

- ***Abstract class MyAbstractList***

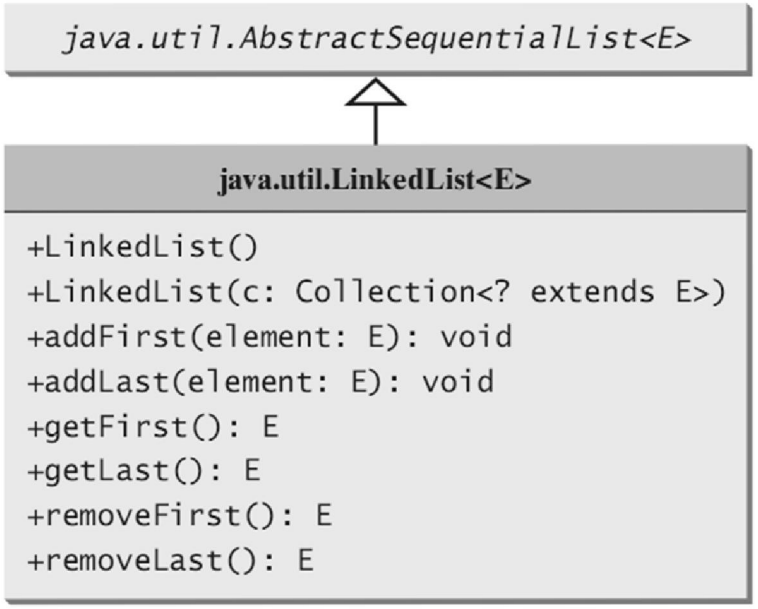
```
public abstract class MyAbstractList<E> implements MyList<E> {

  protected int size = 0; // The size of the list

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

  protected MyAbstractList(E[] objects) { /** Create a list from an array of 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 doesn't contain any 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;
  }
}
```

LinkedList (1)

- Stores elements in a *linked list*
- Which of the two classes, *ArrayList* or *LinkedList*, you use depends on your specific needs:
  - If you need to support random access through an index without inserting or removing elements at the beginning of the list, *ArrayList* offers the most efficient collection
  - If, however, your application requires the insertion or deletion of elements at the beginning of the list, you should choose *LinkedList*
- A list can grow or shrink dynamically
- Once it is created, an array is fixed. If your application does not require the insertion or deletion of elements, an array is the most efficient data structure

LinkedList (2)

- *LinkedList* is a linked list implementation of the List interface
- In addition to implementing the List interface, this class provides the methods for retrieving, inserting, and removing elements from both ends of the list



Creates a default empty linked list.	
Creates a linked list from an existing collection.	
Adds the element to the head of this list.	
Adds the element to the tail of this list.	
Returns the first element from this list.	
Returns the last element from this list.	
Returns and removes the first element from this list.	
Returns and removes the last element from this list.	

LinkedList (3)

```
Linked List Content: [Item1, Item5, Item3, Item6, Item2]
LinkedList Content: [First Item, Item1, Item5, Item3, Item6, Item2, Last Item]
First element: First Item
First element after update: Changed first item
```

- *Example:*

```
import java.util.*;
public class LinkedListExample {
    public static void main(String args[]) {
        /* Linked List Declaration */
        LinkedList<String> linkedlist = new LinkedList<String>();
        linkedlist.addAll(Arrays.asList
                          "Item1","Item5","Item3","Item6","Item2"));
        System.out.println("Linked List Content: " + linkedlist);

        /*Add First and Last Element*/
        linkedlist.addFirst("First Item");
        linkedlist.addLast("Last Item");
        System.out.println("LinkedList Content: " + linkedlist);

        /*This is how to get and set Values*/
        Object firstvar = linkedlist.get(0);
        System.out.println("First element: " + firstvar);
        linkedlist.set(0, "Changed first item");
        Object firstvar2 = linkedlist.get(0);
        System.out.println("First element after update: " +firstvar2);
```

LinkedList (4)

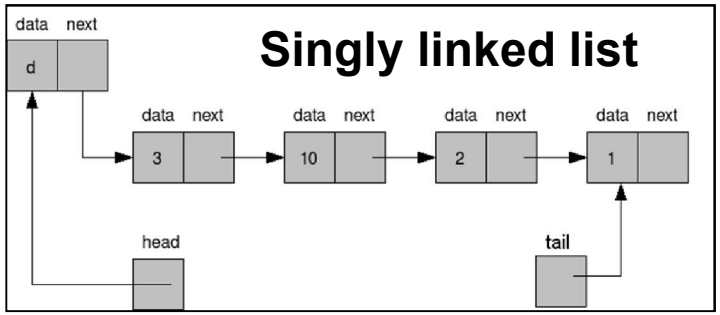
```
[Item1, Item5, Item3, Item6, Item2]
[Newly added item, Item1, Item3, Item6, Item2]
Display the linked list forward:
Newly added item Item1 Item3 Item6 Item2
Display the linked list backward:
Item2 Item6 Item3 Item1 Newly added item
```

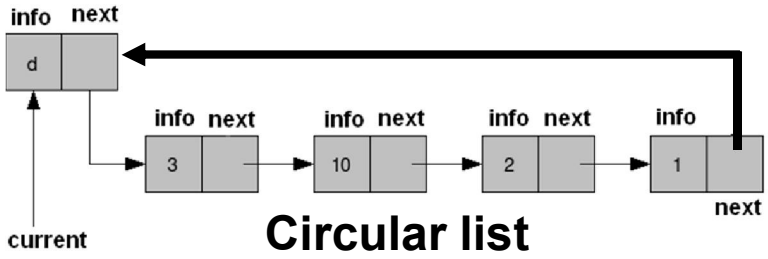
- *Example (Cont'd):*

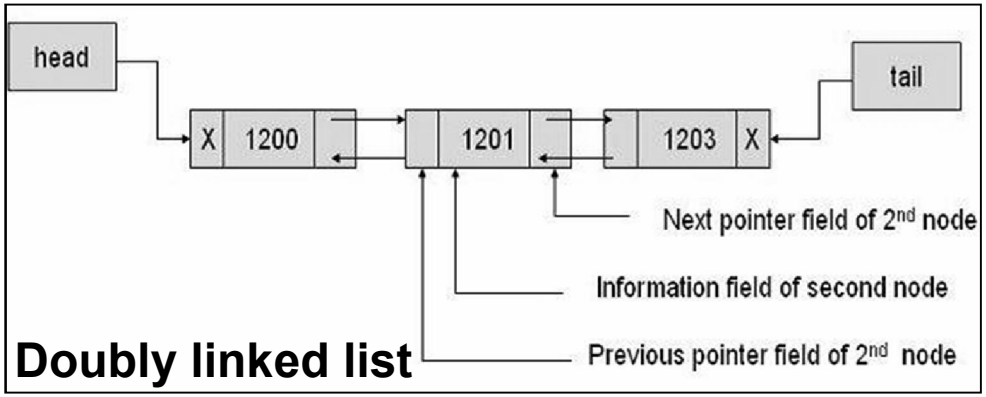
```
/*Remove first and last element*/
linkedlist.removeFirst();
linkedlist.removeLast();
System.out.printlnlinkedlist);
/* Add to a Position and remove from a position*/
linkedlist.add(0, "Newly added item");
linkedlist.remove(2);
System.out.println(linkedlist);
/* Display list forward and backward */
System.out.println("Display the linked list forward:");
ListIterator<String> listIterator = linkedlist.listIterator();
while (listIterator.hasNext())
        System.out.print(listIterator.next() + " ");
System.out.println();

System.out.println("Display the linked list backward:");
listIterator = linkedlist.listIterator(linkedlist.size());
while (listIterator.hasPrevious())
    System.out.print(listIterator.previous() + " ");
}}
```

LinkedList (5)

- Different Types of Lists:
  - Singly linked lists
  - Doubly linked lists
  - Circular lists
  - Skip lists, self-organizing lists, …: Not covered here

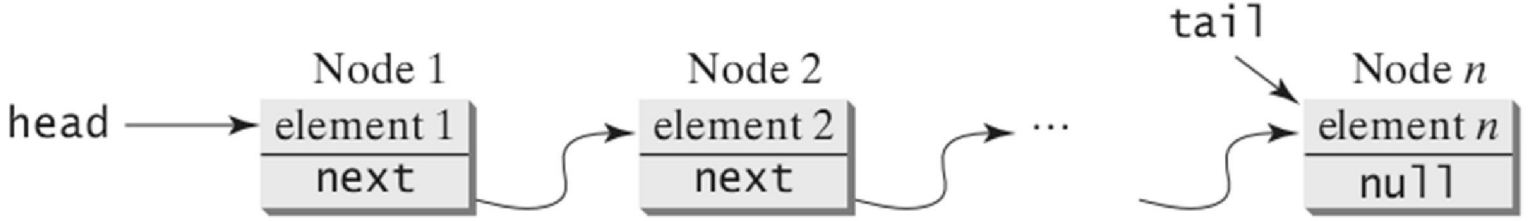






Singly LinkedList (1)

- *Singly Linked List Implementation:*
  - In a linked list, each element is contained in an object, called node
  - When a new element is added, a node is created to contain it
  - Each node is linked to its next neighbor

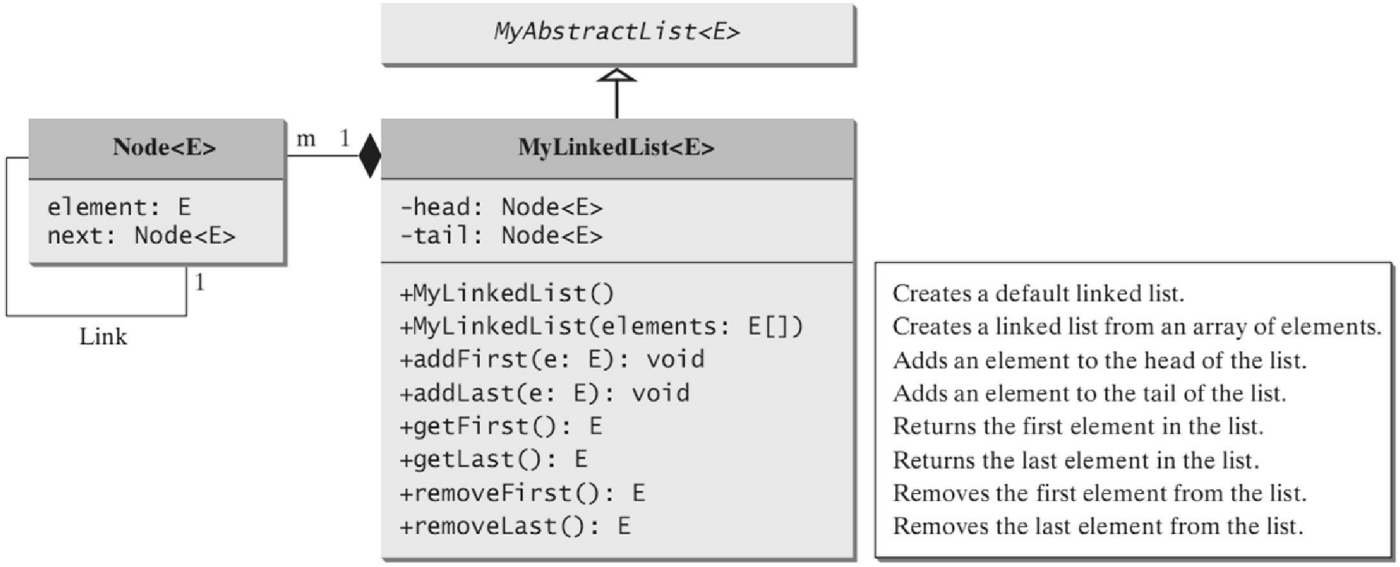


- Example: Creating a linked list using Node

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

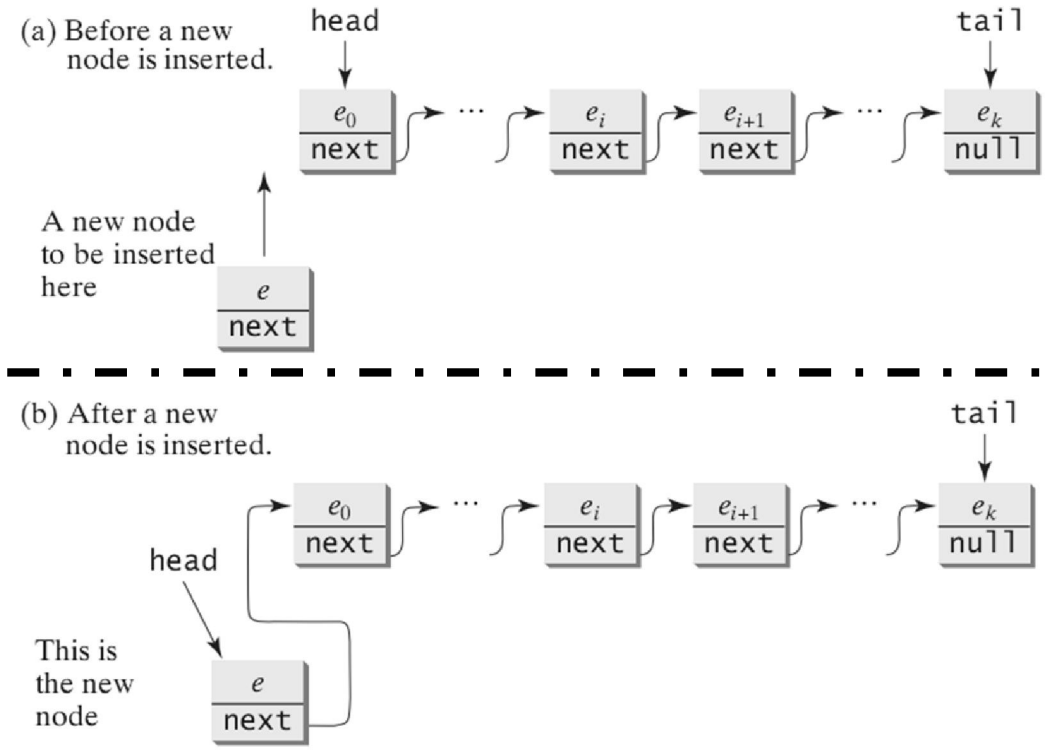
Singly LinkedList (2)

- *Singly Linked List Implementation (Cont'd):*



Singly LinkedList (3)

- *Implementing addirst:*



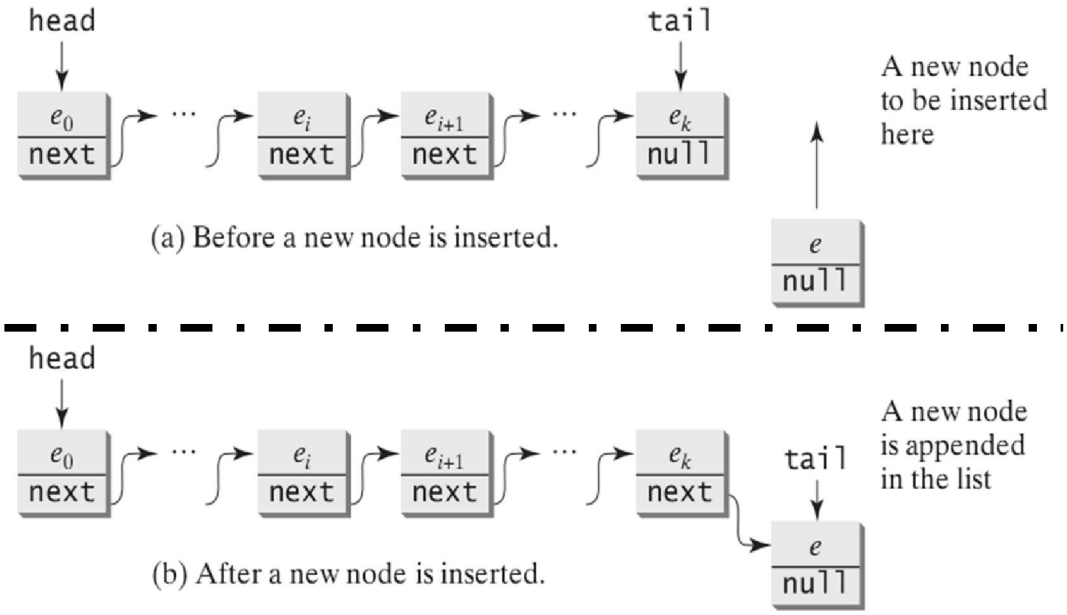

```
1  public void addFirst(E e) {
2    Node<E> newNode = new Node<>(e); // Create a new node
3    newNode.next = head; // link the new node with the head
4    head = newNode; // head points to the new node
5    size++; // Increase list size
6
7    if (tail == null) // The new node is the only node in list
8      tail = head;
9  }
```

create a node
link with head
head to new node
increase size
was empty?

Complexity: O(1)

Singly LinkedList (4)

- *Implementing addLast:*




```
1  public void addLast(E e) {
2    Node<E> newNode = new Node<>(e); // Create a new node for e
3
4    if (tail == null) {
5      head = tail = newNode; // The only node in list
6    }
7    else {
8      tail.next = newNode; // Link the new node with the last node
9      tail = tail.next; // tail now points to the last node
10   }
11
12   size++; // Increase size
13 }
```

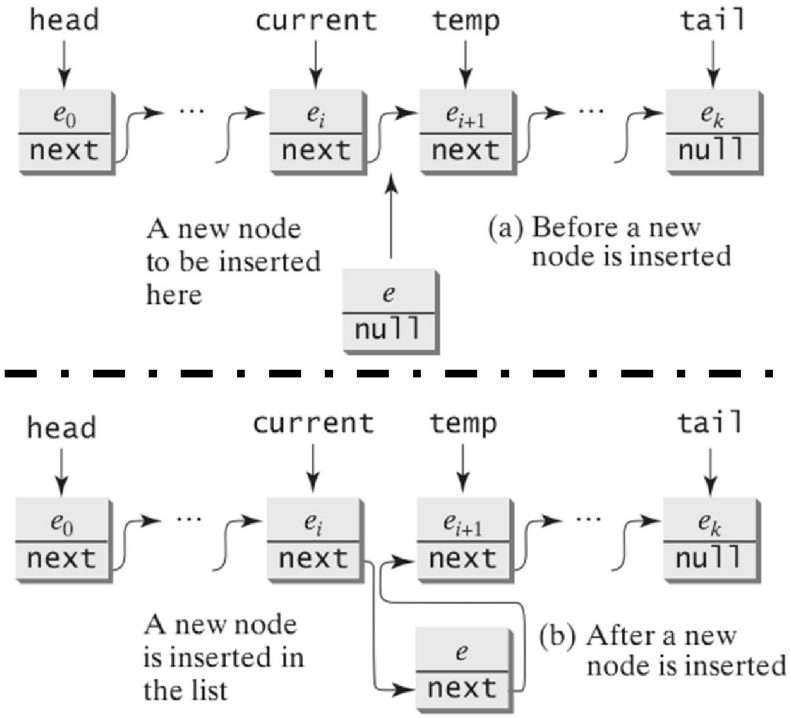
create a node

increase size

Complexity: O(1)

Singly LinkedList (5)

- *Implementing add:*

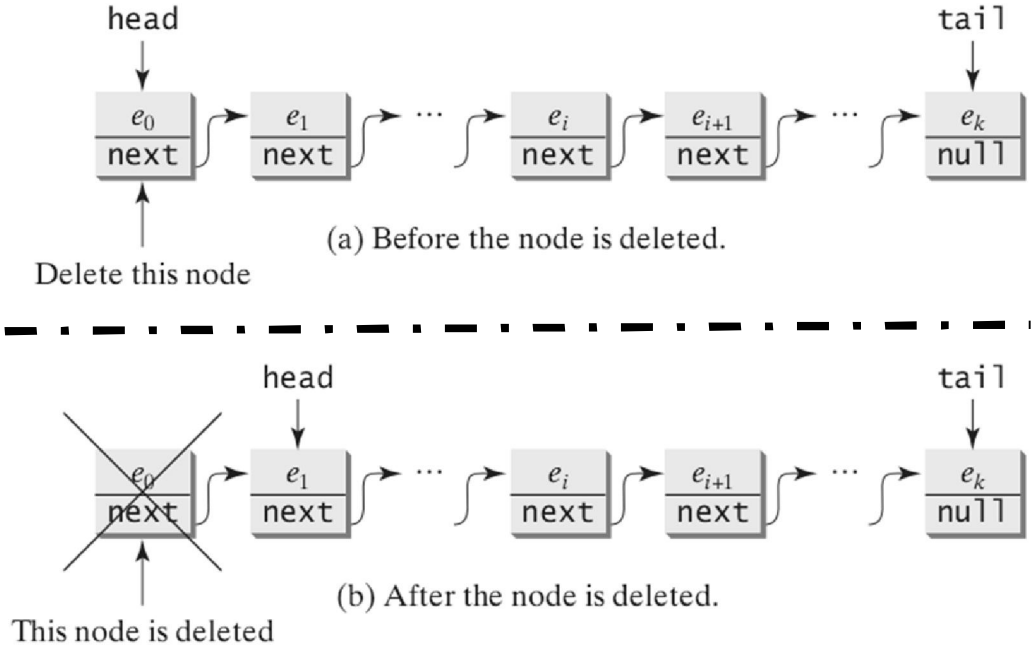



```
1  public void add(int index, E e) {
2    if (index == 0) addFirst(e); // Insert first
3    else if (index >= size) addLast(e); // Insert last
4    else { // Insert in the middle
5      Node<E> current = head;
6      for (int i = 1; i < index; i++)
7        current = current.next;
8      Node<E> temp = current.next;
9      current.next = new Node<E>(e);
10     (current.next).next = temp;
11     size++;
12   }
13 }
```

insert first
insert last
create a node
increase size

Complexity: O(n)

Singly LinkedList (6)



- *Implementing removeFirst:*

```
1  public E removeFirst() {
2    if (size == 0) return null; // Nothing to delete
3    else {
4      Node<E> temp = head; // Keep the first node temporarily
5      head = head.next; // Move head to point to next node
6      size--; // Reduce size by 1
7      if (head == null) tail = null; // List becomes empty
8      return temp.element; // Return the deleted element
9    }
10 }
```

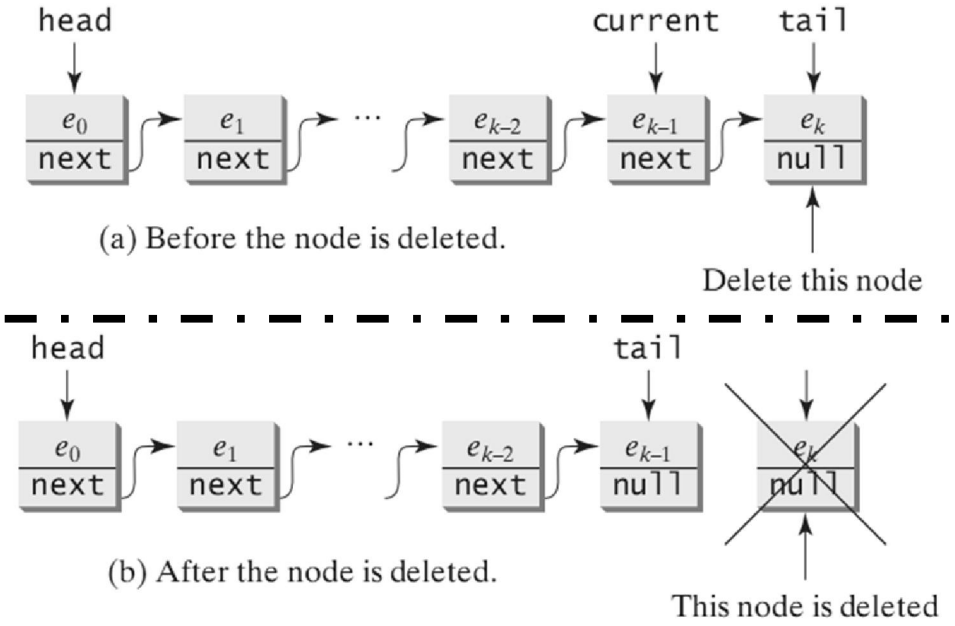
nothing to remove

keep old head
new head
decrease size
destroy the node

Complexity: O(1)

Singly LinkedList (7)

- *Implementing removeLast:*

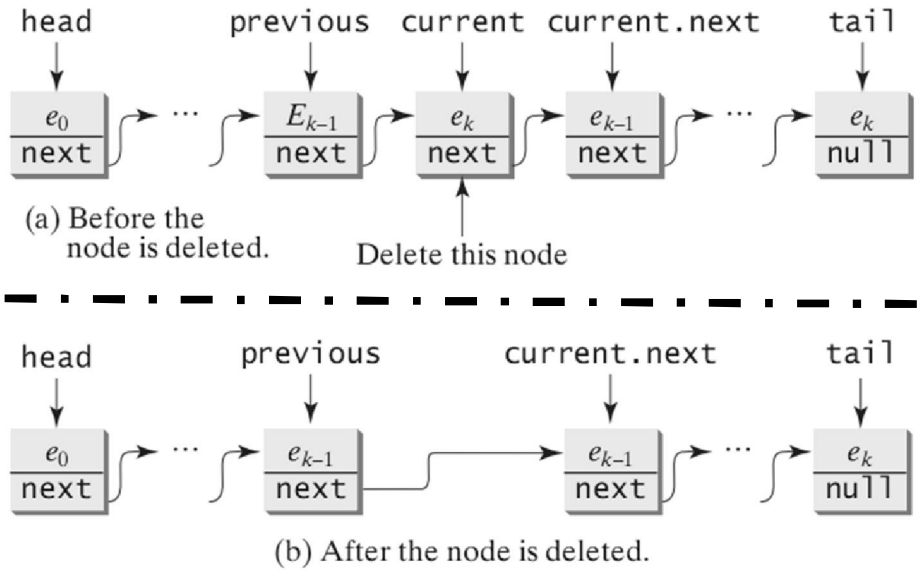



```
                          1  public E removeLast() {
empty?                    2    if (size == 0) return null;
size 1?                   3    else if (size == 1) {
                          4      Node<E> temp = head;
head and tail null        5      head = tail = null;
size is 0                 6      size = 0;
return element            7      return temp.element;
                          8    }
size > 1                  9    else {
                         10      Node<E> current = head;
                         11
                         12      for (int i = 0; i < size - 2; i++)
                         13        current = current.next;
                         14
                         15      Node<E> temp = tail;
move tail                16      tail = current;
                         17      tail.next = null;
reduce size              18      size--;
return element           19      return temp.element;
                         20    }
                         21  }
```

Complexity: O(n)

Singly LinkedList (8)

- *Implementing remove:*
- *Do Group Activity 3*



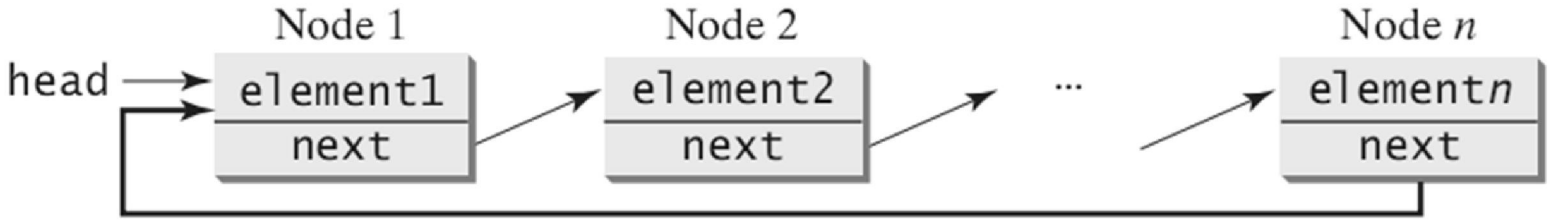

```
1  public E remove(int index) {
2    if (index < 0 || index >= size) return null; // Out of range
3    else if (index == 0) return removeFirst(); // Remove first
4    else if (index == size - 1) return removeLast(); // Remove last
5    else {
6      Node<E> previous = head;
7
8      for (int i = 1; i < index; i++) {
9        previous = previous.next;
10     }
11
12     Node<E> current = previous.next;
13     previous.next = current.next;
14     size--;
15     return current.element;
16   }
17 }
```

out of range
remove first
remove last
locate previous
locate current
remove from list
reduce size
return element

Complexity: O(n)

Circular LinkedList (1)

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



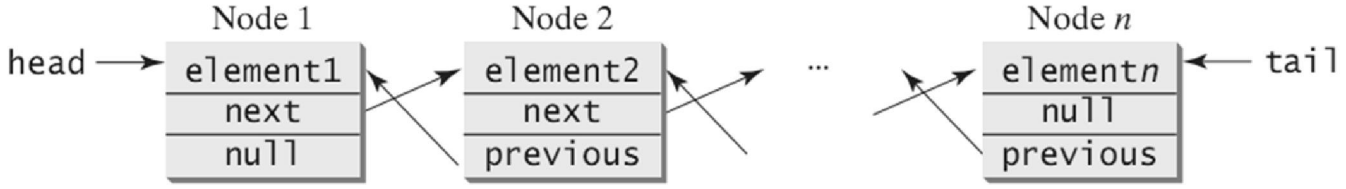
- tail is not needed
- head points to the current node in the list
- Insertion and deletion take place at the current node
- Example of application:
  - Operating System that serves multiple users in a timesharing fashion
  - The system picks a user from a circular list and grants a small amount of CPU time, then moves on to the next user in the list
- Group Activity 4

Doubly LinkedList (1)

- *Time Complexities for Methods in MyArrayList and MyLinkedList*

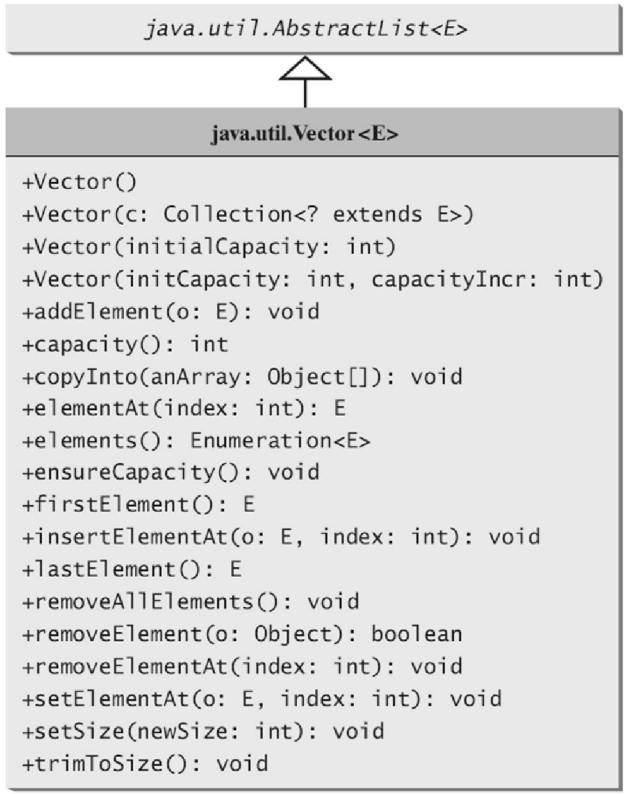
Methods	*MyArrayList/ArrayList*	*MyLinkedList/LinkedList*
`add(e: E)`	$O(1)$	$O(1)$
`add(index: int, e: E)`	$O(n)$	$O(n)$
`clear()`	$O(1)$	$O(1)$
`contains(e: E)`	$O(n)$	$O(n)$
`get(index: int)`	$O(1)$	$O(n)$
`indexOf(e: E)`	$O(n)$	$O(n)$
`isEmpty()`	$O(1)$	$O(1)$
`lastIndexOf(e: E)`	$O(n)$	$O(n)$
`remove(e: E)`	$O(n)$	$O(n)$
`size()`	$O(1)$	$O(1)$
`remove(index: int)`	$O(n)$	$O(n)$
`set(index: int, e: E)`	$O(n)$	$O(n)$
`addFirst(e: E)`	$O(n)$	$O(1)$
`removeFirst()`	$O(n)$	$O(1)$

- **Group Activity 5:** Doubly Linked List implementation



Vector

- Vector is a subclass of AbstractList
- Vector is the same as ArrayList, except that it contains ***synchronized methods*** for accessing and modifying the vector
- Synchronized methods can prevent data corruption when a vector is accessed and modified by two or more threads concurrently
- Use the ArrayList class if you don't need synchronization
- ArrayList works much faster than Vector
- ArrayList Vs Vector



Creates a default empty vector with initial capacity 10.
Creates a vector from an existing collection.
Creates a vector with the specified initial capacity.
Creates a vector with the specified initial capacity and increment.
Appends the element to the end of this vector.
Returns the current capacity of this vector.
Copies the elements in this vector to the array.
Returns the object at the specified index.
Returns an enumeration of this vector.
Increases the capacity of this vector.
Returns the first element in this vector.
Inserts o into this vector at the specified index.
Returns the last element in this vector.
Removes all the elements in this vector.
Removes the first matching element in this vector.
Removes the element at the specified index.
Sets a new element at the specified index.
Sets a new size in this vector.
Trims the capacity of this vector to its size.