

Linked List

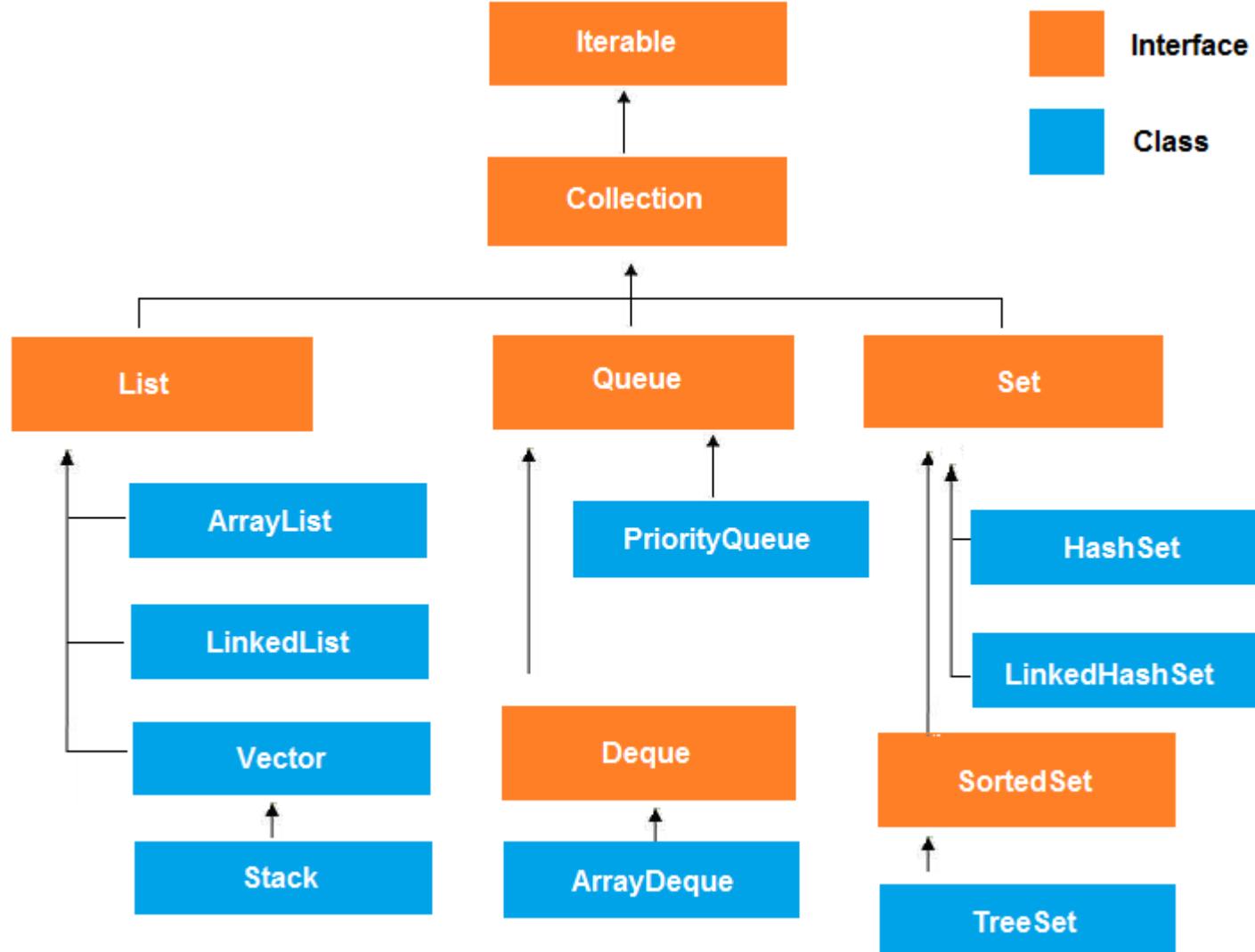
WIA1002/WIB1002
Data Structure

Content

Java Collection Framework hierarchy

A *collection* is a container object that holds a group of objects, often referred to as *elements*.

Java Collection Framework hierarchy



«interface»
java.lang.Iterable<E>

+iterator(): Iterator<E>



«interface»
java.util.Collection<E>

+add(o: E): boolean
+addAll(c: Collection<? extends E>): boolean
+clear(): void
+contains(o: Object): boolean
+containsAll(c: Collection<?>): boolean
+equals(o: Object): boolean
+hashCode(): int
+isEmpty(): boolean
+remove(o: Object): boolean
+removeAll(c: Collection<?>): boolean
+retainAll(c: Collection<?>): boolean
+size(): int
+toArray(): Object[]

The Collection Interface

Returns an iterator for the elements in this collection.

The **Collection** interface is the root interface for manipulating a collection of objects.

Adds a new element o to this collection.

Adds all the elements in the collection c to this collection.

Removes all the elements from this collection.

Returns true if this collection contains the element o.

Returns true if this collection contains all the elements in c.

Returns true if this collection is equal to another collection o.

Returns the hash code for this collection.

Returns true if this collection contains no elements.

Removes the element o from this collection.

Removes all the elements in c from this collection.

Retains the elements that are both in c and in this collection.

Returns the number of elements in this collection.

Returns an array of Object for the elements in this collection.

«interface»
java.util.Iterator<E>

+hasNext(): boolean
+next(): E
+remove(): void

Returns true if this iterator has more elements to traverse.

Returns the next element from this iterator.

Removes the last element obtained using the next method.

Lists

- A list is a popular abstract data type that stores 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.
- The common operations on a list are :
 - . Retrieve an element from this list.
 - . Insert a new element to 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

1. Using an array to store the elements.
 - The array is dynamically created.
 - If array capacity is exceeded, create a new larger array and copy all the elements from the current array to the new array.
2. 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.

Choose array or linked list to implement a list?

- Use an array
 - `get(int index)` and `set(int index, Object o)` through an index and `add(Object o)` for adding an element at the end of the list are efficient.
 - `add(int index, Object o)` and `remove(int index)` are inefficient - shift potentially large number of elements.
- Using linked list
 - improve efficiency for adding and removing an element anywhere in a list.

Introducing Linked Lists

- ♦ Think of each element in a linked list as being an individual piece in a child's pop chain. To form a chain, insert the connector into the back of the next piece



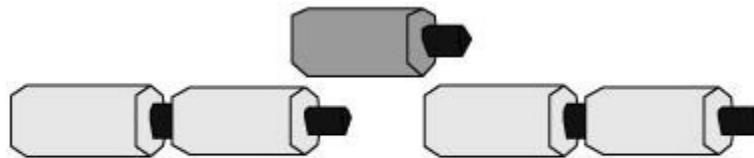
Individual Piece



Pop Chain

Introducing Linked Lists

- Inserting a new piece into the chain involves merely breaking a connection and reconnecting the chain at both ends of the new piece.



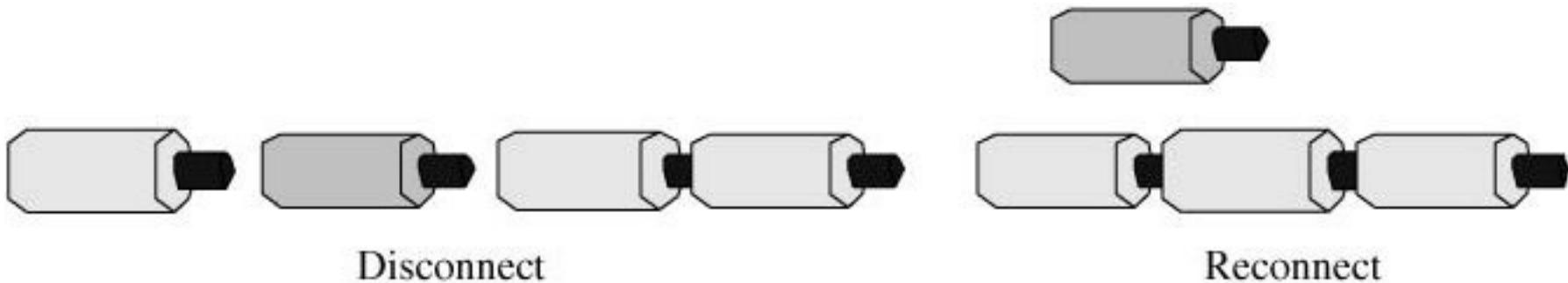
Disconnect



Reconnect

Introducing Linked Lists

- Removal of a piece from anywhere in the chain requires breaking its two connections, removing the piece, and then reconnecting the chain.

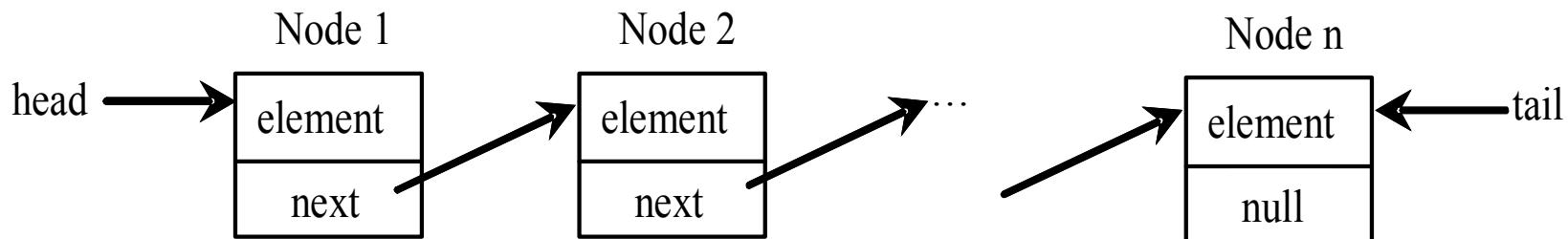


Introducing Linked Lists

- Inserting and deleting an element is a local operation and requires updating only the links adjacent to the element. The other elements in the list are not affected.

Nodes in Linked Lists

- A linked list consists of nodes.
- Each node contains an element, and each node is linked to its next neighbor.
- A node with its two fields can reside anywhere in memory.



```
class Node<E> {  
    E element;      //contains the element  
    Node<E> next; // a reference to the next node  
  
    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<>("Chicago");  
tail = head;
```

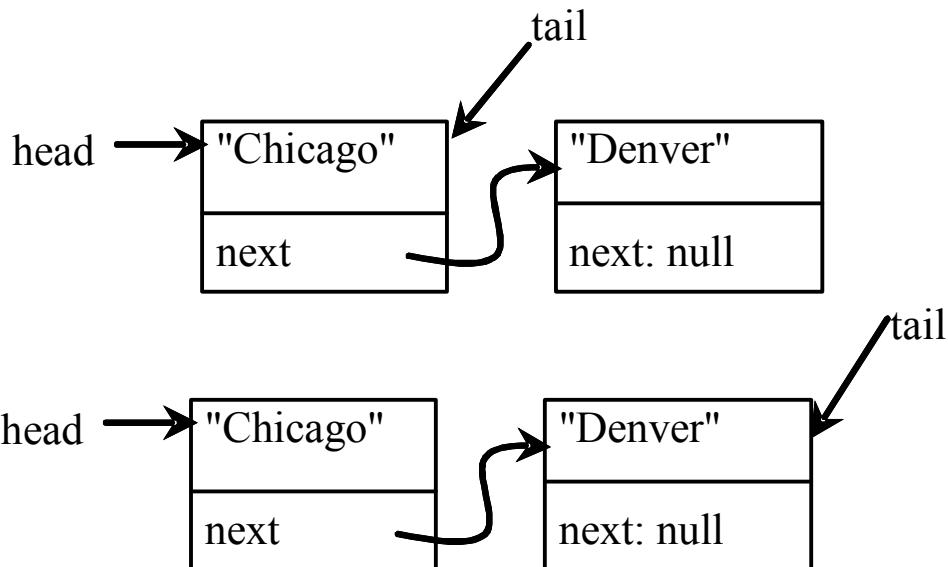
After the first node is inserted



Adding Three Nodes, cont.

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

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

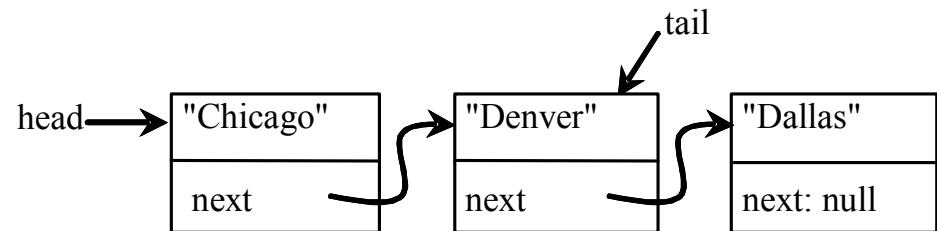


```
tail = tail.next();
```

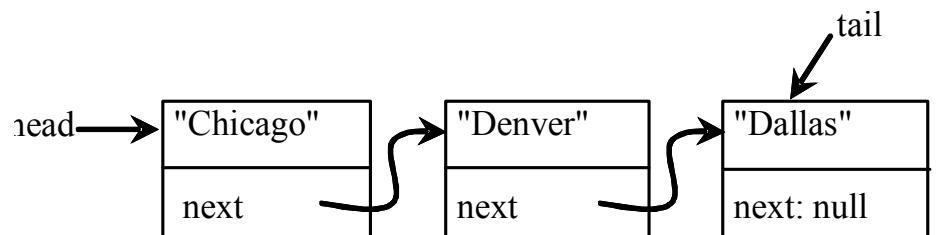
Adding Three Nodes, cont.

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

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



```
tail = tail.next();
```



Traversing All Elements in the List

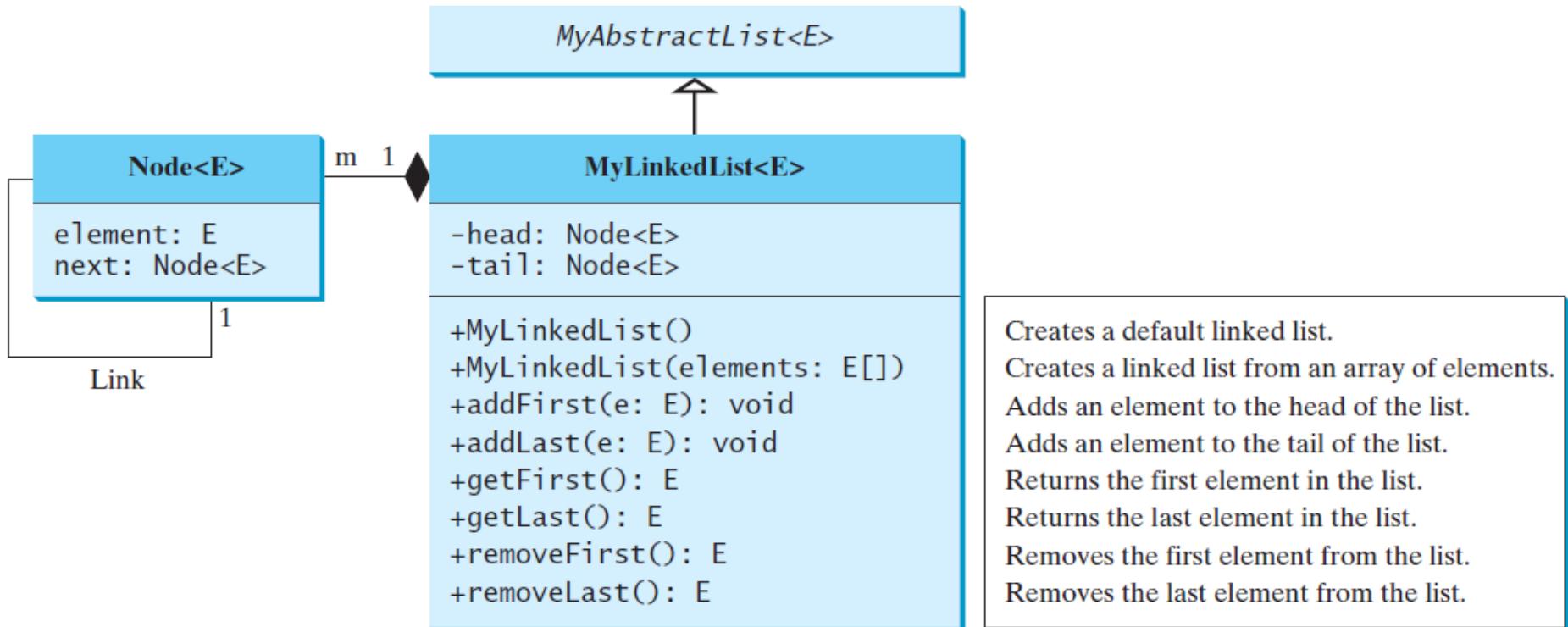
Each node contains the element and a data field named *next* that points to the next node.

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.

Loop to traverse all the nodes in the list:

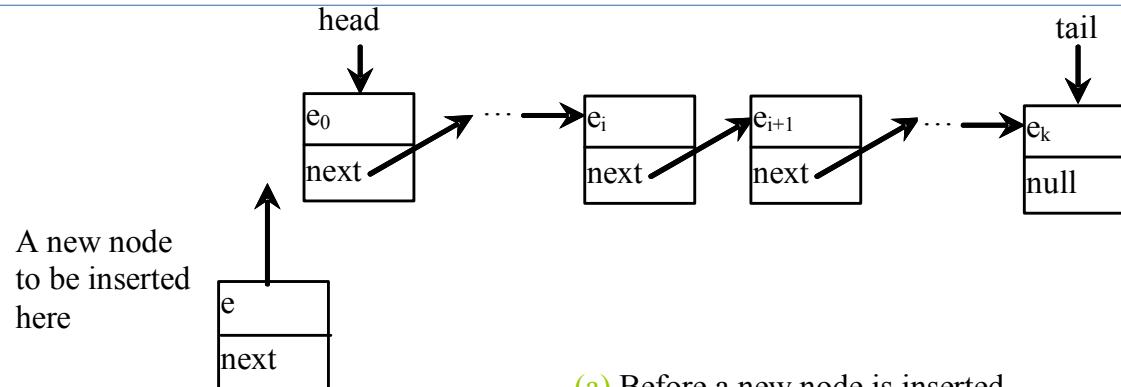
```
Node<E> current = head;  
while (current != null) {  
    System.out.println(current.element);  
    current = current.next;  
    //continuously moving forward  
}
```

MyLinkedList

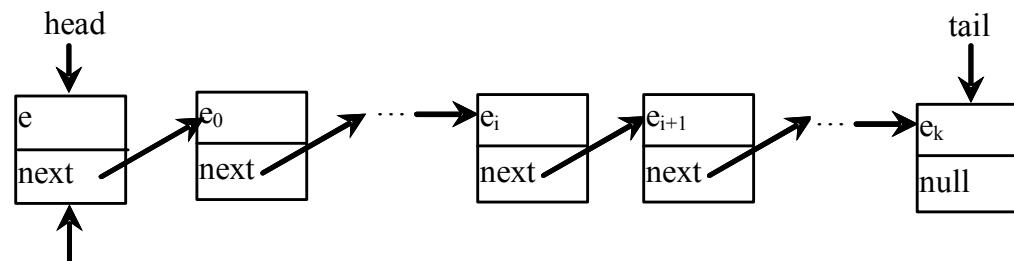


Implementing addFirst(E e)

```
public void addFirst(E e) {  
    Node<E> newNode = new Node<>(e);  
    newNode.next = head; //create pointer to current head  
    head = newNode; //new node created & assigned to new head  
    size++; //increase size  
    if (tail == null) //no node exists  
        tail = head;  
}
```



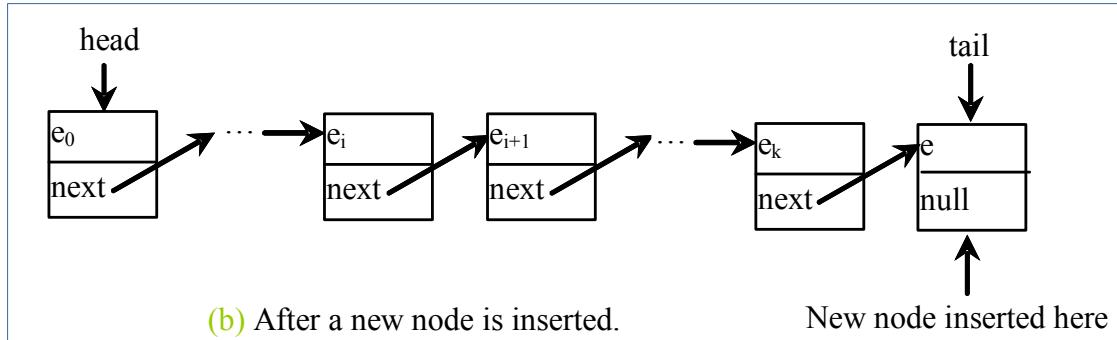
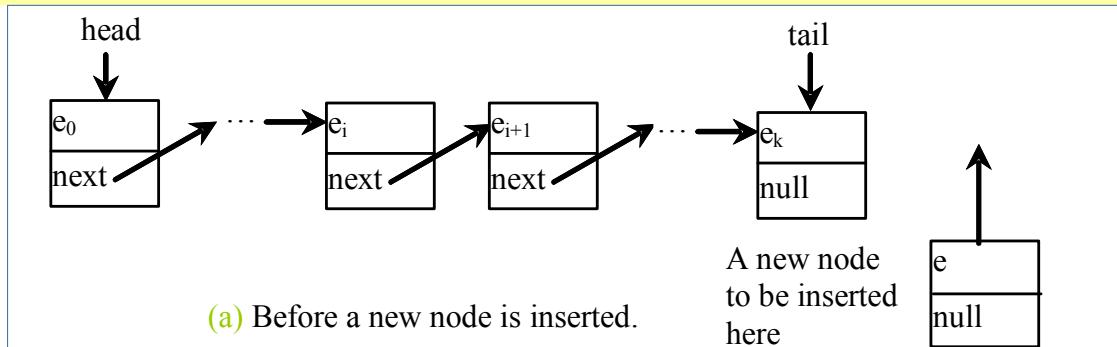
(a) Before a new node is inserted.



(b) After a new node is inserted.

Implementing addLast(E e)

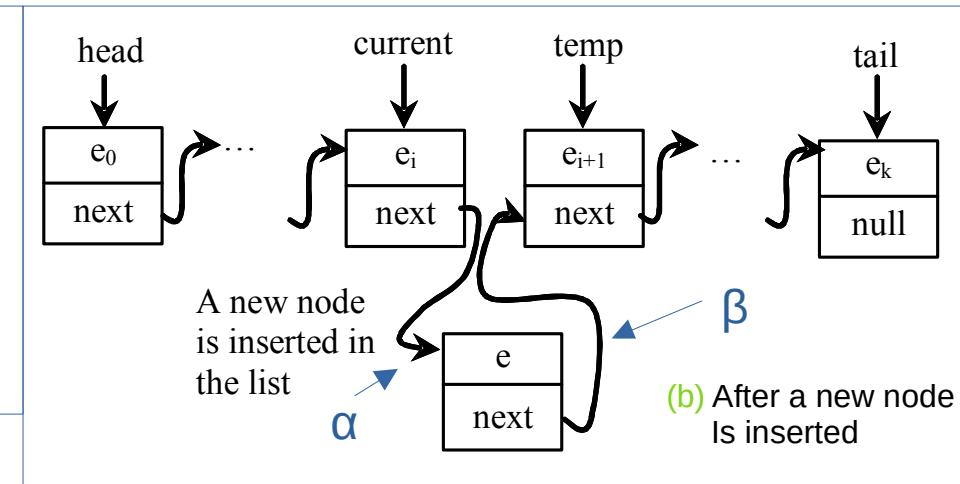
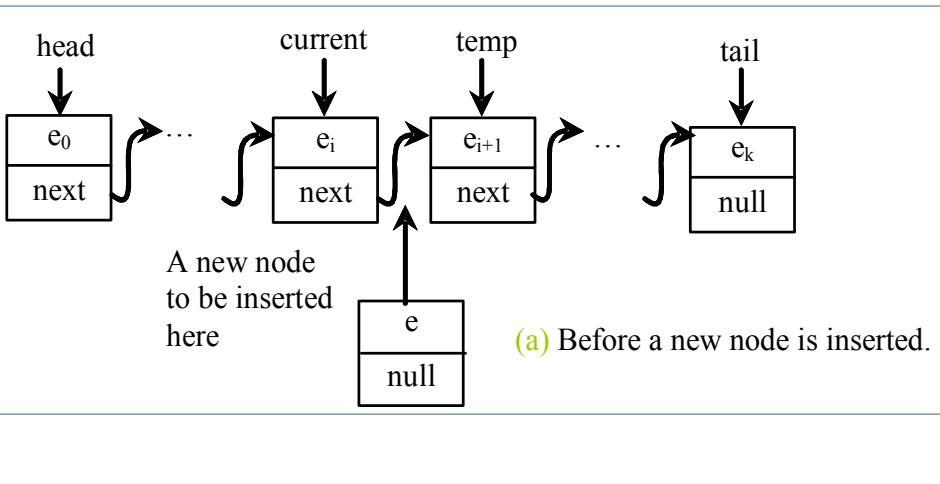
```
public void addLast(E e) {  
    if (tail == null) { //no node exist  
        head = tail = new Node<>(e);  
    }  
    else {  
        tail.next = new Node<>(e); //tail.next point to new Node  
        tail = tail.next; //new tail updated from tail.next  
    }  
    size++;  
}
```



Implementing add(int index, E e)

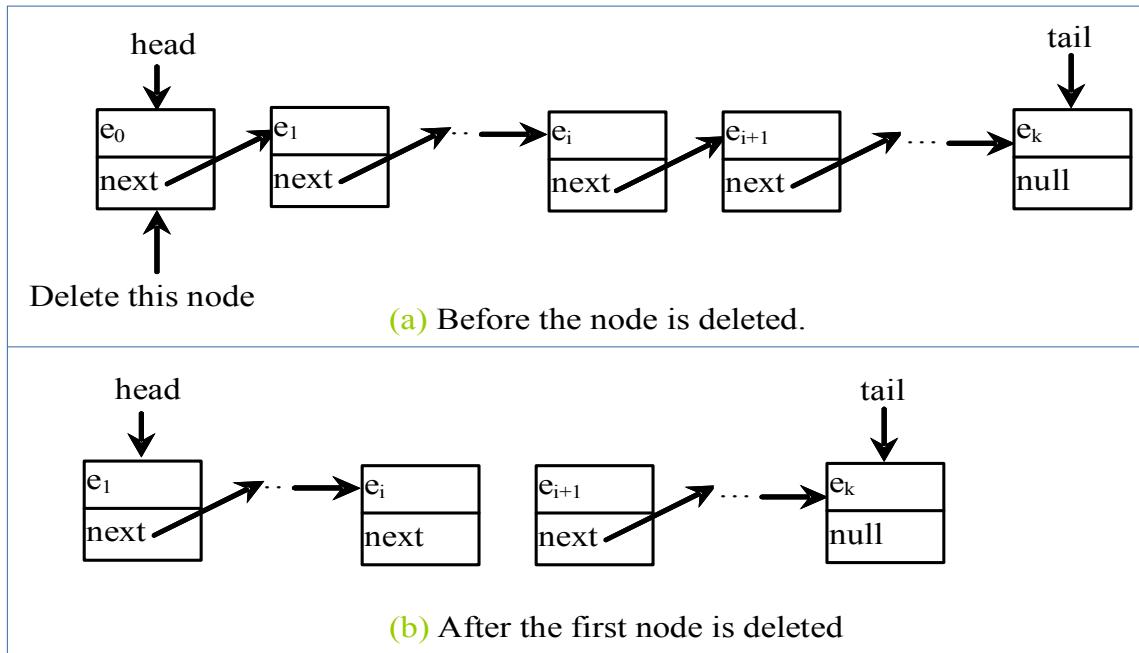
```

1  public void add(int index, E e) {
2      if (index == 0) addFirst(e);      //since requested to add at index 0
3      else if (index >= size) addLast(e); //since requested to add at index=size
4      else {
5          Node<E> current = head;      //set head to be a current node
6          for (int i = 1; i < index; i++) //traverse & stop before requested index
7              current = current.next;
8          Node<E> temp = current.next; //hold reference current.next
9          current.next = new Node<>(e); //current.next point to new node (refer α)
10         (current.next).next = temp;   //get the reference from temp (refer β)
11         size++;
12     }
13 }
```



Implementing removeFirst()

```
public E removeFirst() {  
    if (size == 0) return null; // no node then return null  
    else {  
        Node<E> temp = head; // copy head to temp node before delete  
        head = head.next; //set new head  
        size--; //reduce size  
        if (head == null) tail = null; //in case of head=null  
        return temp.element; //to know what we delete  
    }  
}
```

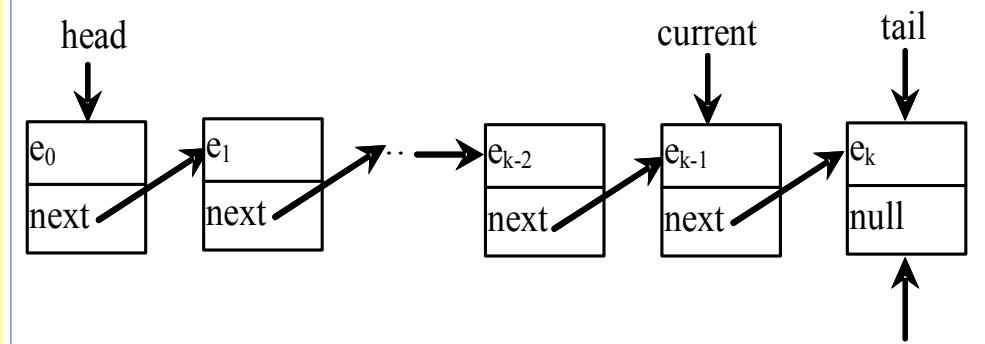


```

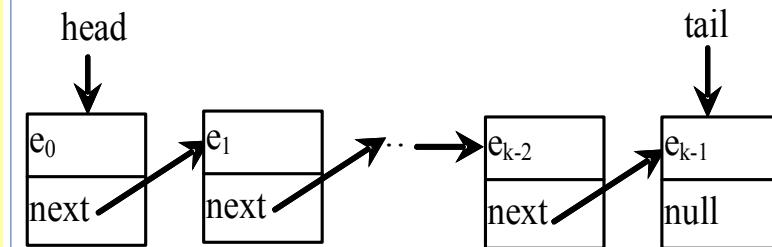
public E removeLast() {
    if (size == 0) return null;
    else if (size == 1) //only 1 node
    {
        Node<E> temp = head;
        head = tail = null;
        //reset to know
        size = 0;
        return temp.element;
        //to know what we delete
    }
    else
    {
        Node<E> current = head;
        for (int i = 0; i < size - 2; i++)
            current = current.next;
        //stop 1 node before tail
        Node<E> temp = tail;
        //copy tail to temp b4 delete
        tail = current;
        //current become tail
        tail.next = null;
        //reset the next for tail
        // to be null
        size--;
        return temp.element;
    }
}

```

Implementing removeLast()



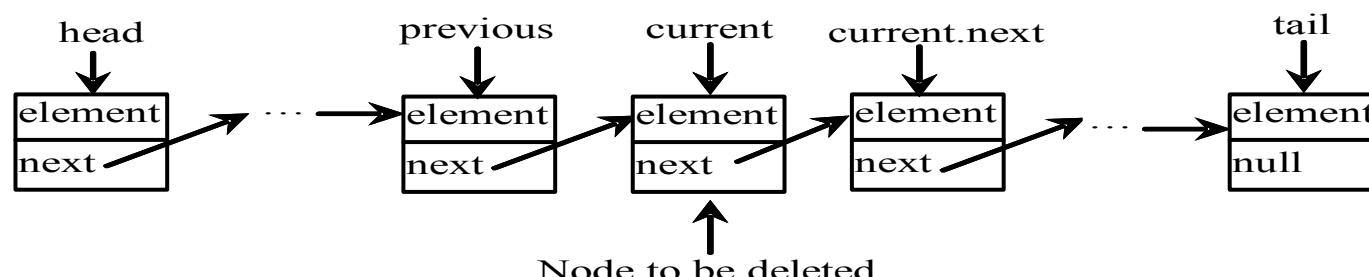
(a) Before the node is deleted.



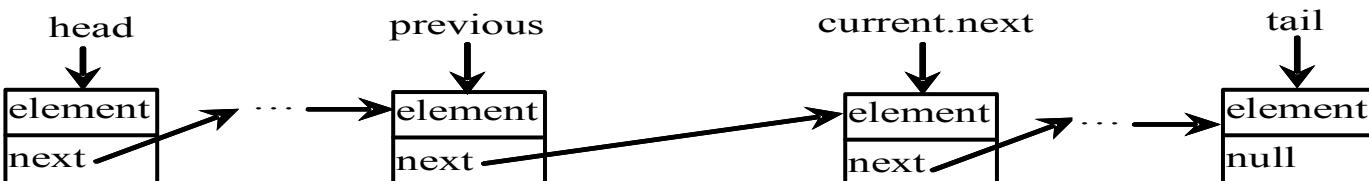
(b) After the last node is deleted

Implementing remove(int index)

```
public E remove(int index) {  
    if (index < 0 || index >= size) return null; // to delete index of node not in range  
    else if (index == 0) return removeFirst(); //call removeFirst  
    else if (index == size - 1) return removeLast(); //call removeLast  
    else {  
        Node<E> previous = head; //Set head to be previous  
        for (int i = 1; i < index; i++) {  
            previous = previous.next; // stop before index that want to be deleted  
        }  
        Node<E> current = previous.next; //copy previous.next to current  
        previous.next = current.next; //set new point to from previous.next to current.next  
        size--; //reduce size  
        return current.element;  
    }  
}
```



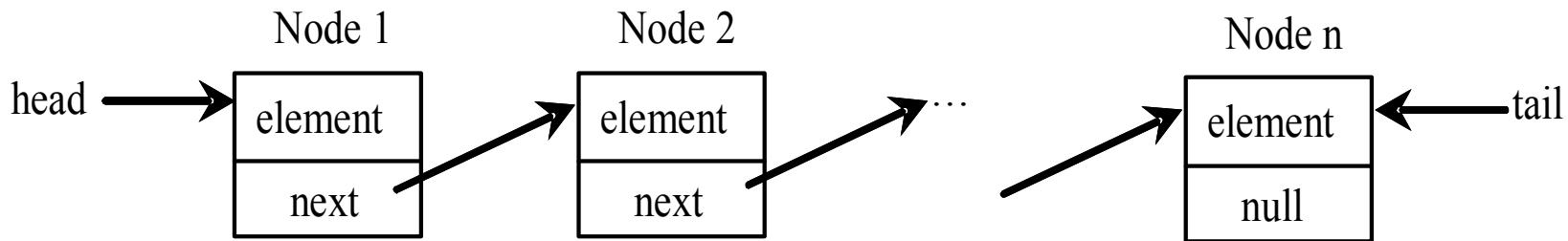
(a) Before the node is deleted.



(b) After the node is deleted.

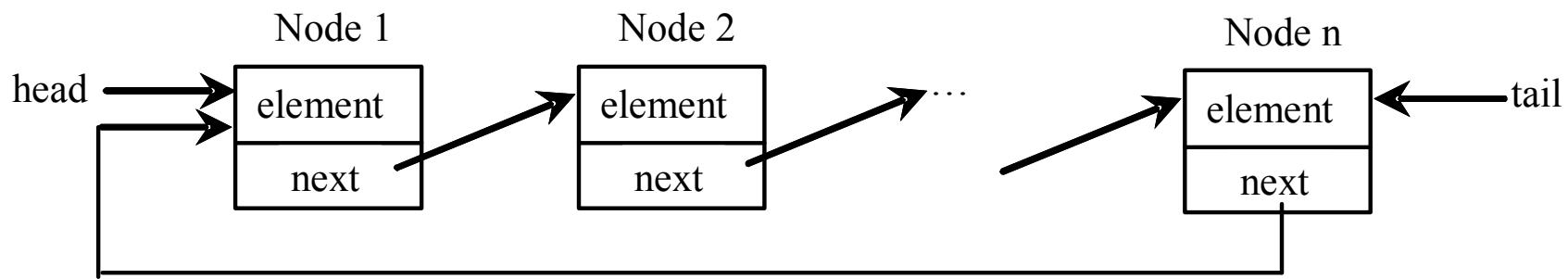
Singly linked list

- ❖ What you have seen so far is singly linked list (contains a pointer to the list's first node, and each node contains a pointer to the next node sequentially.)
- ❖ Is not a direct access structure. It must be accessed sequentially by moving forward one node at a time



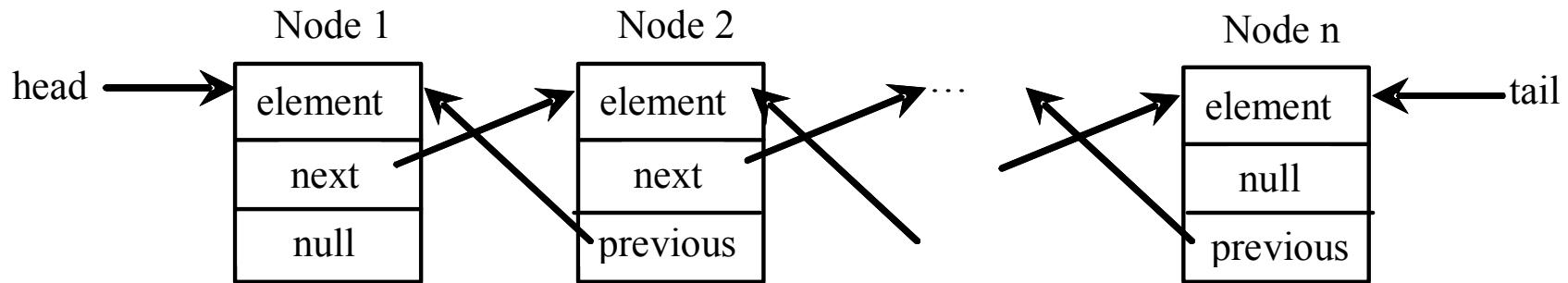
Circular Linked Lists

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**.



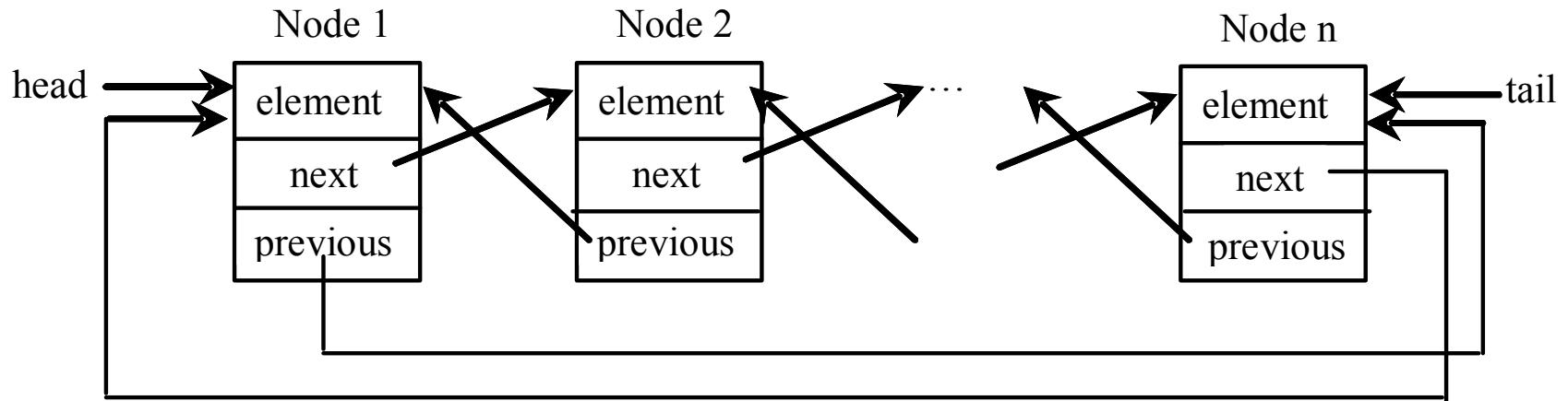
Doubly Linked Lists

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.



References

1. Chapter 20, Liang, Introduction to Java Programming, 10th Edition, Global Edition, Pearson, 2015
2. Chapter 24, Liang, Introduction to Java Programming, 10th Edition, Global Edition, Pearson, 2015