UBC OKANAGAN

COSC 121
Computer Programming II

Implementing Lists, Stacks, and Queues

Part 1/2

Dr. Mostafa Mohamed

Acknowledgement: The slides mainly rely on the textbook of Y. D. Liang titled “Introduction to Java Programming, 10th Ed.”, Pearson Edu. Inc.

Outline

Today:

- Implementing `LinkedLists`

Next lecture:

- Continue Implementing `LinkedLists`
- Implementing `ArrayLists`
- Implementing `Stacks / Queues`

Two Ways to Implement Lists

1) Using linked lists (we will create **MyLinkedList** class**)**

- 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.

2) Using arrays (we will create **MyArrayList** class**)**.

- 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.

We shall use an abstract class and interface to define the commonalities between MyArrayList and MyLinkedList.

Why learn implementing lists?

Knowing what's happening "**under the hood**" allows you to use the built-in structures more **effectively**.

Helps you implement **custom** data structures that is similar but not identical to built-in classes (which is very likely).

- Do more than what Java built-in classes do
  - Custom design (e.g., *constructor to read elements from a text file*)
  - Custom methods (e.g., *insertAfter, modifyAll, deep clone, invertList, searchSequence*,…)

ArrayLists and LinkedLists are relatively easy data structures to build.

Needed to understand the more complex structures (e.g. in COSC 222).

Useful when using **another** OO language that doesn't provide built-in collection classes.

Allows you to **research** ways to improve current collections.

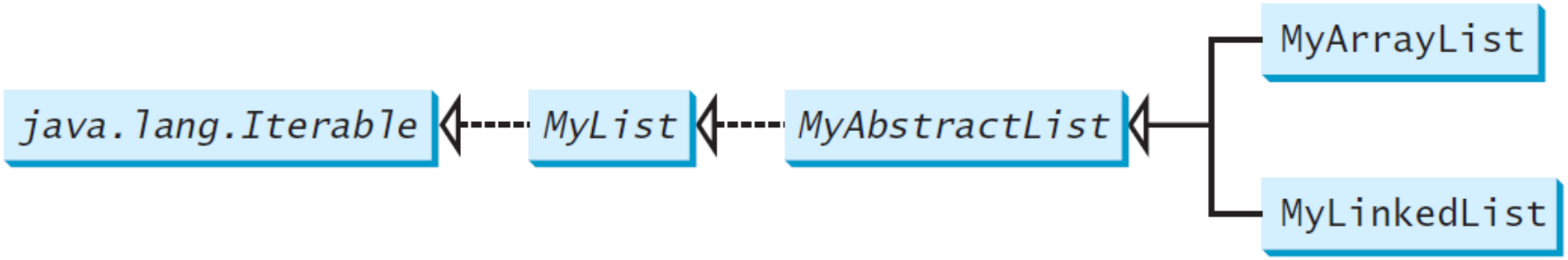
- e.g., head + tail + other pointers (e.g. if you want to insert near middle)

Helps you **practice** your Java skills.

Today's challenge…

Given:

- The code for `MyList` and `MyAbstractList`



Required:

- Implement `MyArrayList`
- Implement `MyLinkedList`

***Note**: what is presented in the following slides is not the optimum implementation, but rather a simpler one to demonstrate how this could be done.*

MyList Interface

```
public interface MyList<E> extends Iterable<E> {
  public void add(E e); // add e at the end
  public void add(int index, E e);

  public boolean remove(E e);
  public E remove(int index);
  public void clear();

  public E get(int index);
  public Object set(int index, E e); //returns new set

  public int indexOf(E e);  //returns -1 if no match
  public int lastIndexOf(E e);
  public boolean contains(E e);
  public boolean isEmpty();

  public int size();
  public java.util.Iterator<E> iterator();
}
```

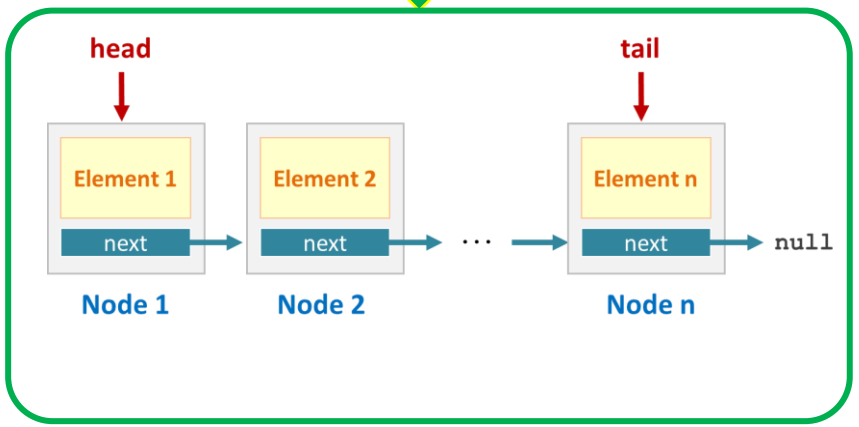
MyAbstractList Class

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

    // Constructors
    protected MyAbstractList() {}
    protected MyAbstractList(E[] objects) {
        for (int i = 0; i < objects.length; i++)
            add(objects[i]);
    }

    // Methods (those with SAME implementation for MyArrayList & MyLinkedList)
    public void add(E e) {
        add(size, e);            //call the other add method
    }
    public boolean remove(E e) {
        if (indexOf(e) >= 0) {  //if the element exists
            remove(indexOf(e)); //call the other remove method
            return true;
        } else
            return false;
    }
    public boolean isEmpty() { return size == 0; }
    public int size() { return size; }
}
```

Implementing Linked Lists

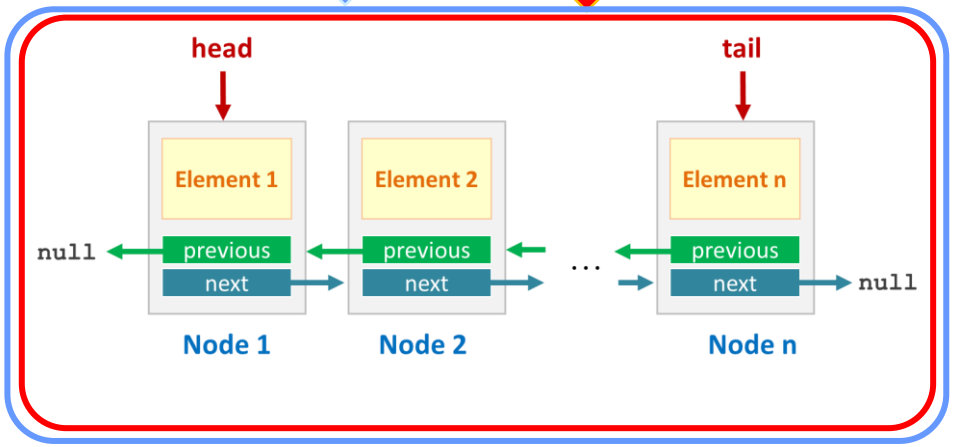
Several types of linked lists…

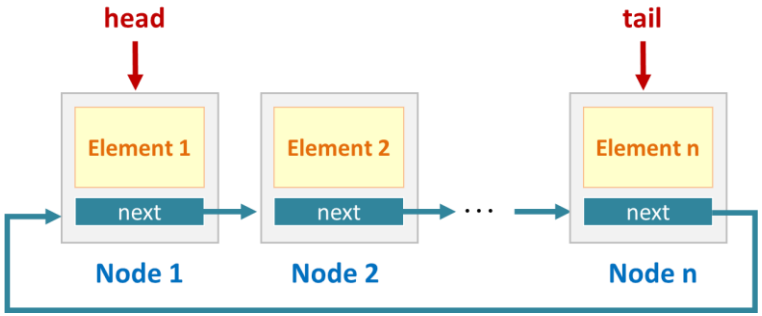
We will first focus on this type for simplicity

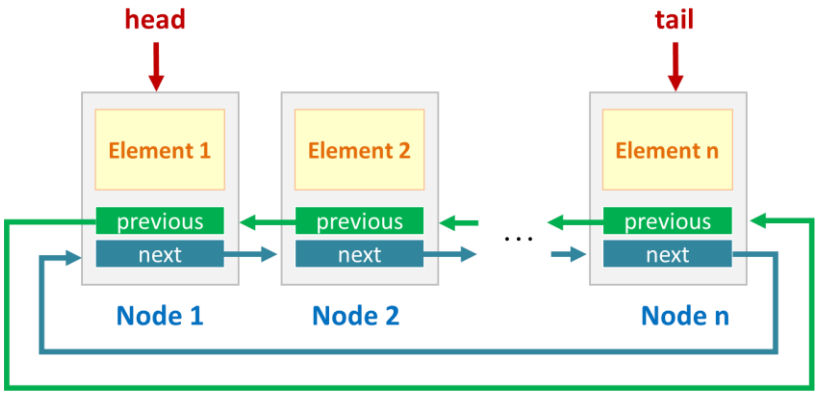


Then we will discuss this type as an exercise

`LinkedList`







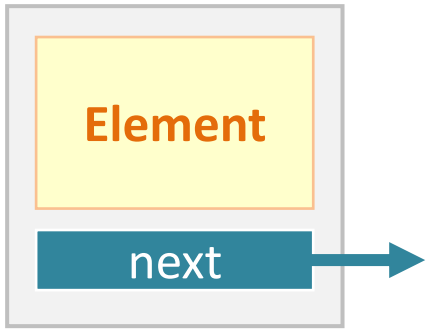
Once you get it, you can implement any type of linked lists

Before we implement MyLinkedList

What is a Node?

- It is an instance of a class that has exactly **two attributes ***:
  - `element`
    - **holds the data** at this node
    - of a generic **type `E`**
  - `next`
    - **refers to the next node** in the linked structure.
    - of the **type `Node<E>`**

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



* It has a third attribute, `previous`, in case of doubly linked lists.

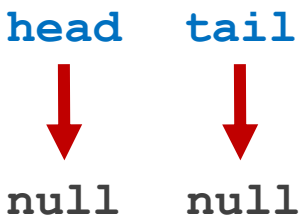
Before we implement MyLinkedList

What is a Linked List?

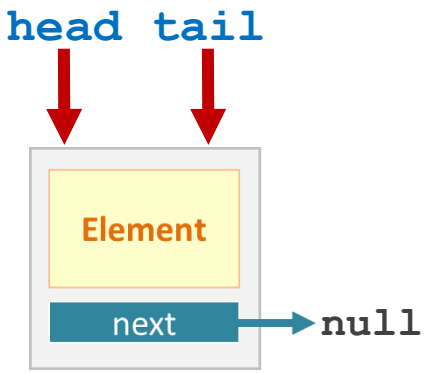
- It is an instance of a class that has **three attributes**:
  - `size`
    - An **integer** that keeps track of the number of nodes
  - `head`
    - points to the first element
    - of the **type** `Node<E>`
  - `tail`
    - points to the last element
    - of the **type** `Node<E>`

When we implement linked lists today, we will need to consider **3 cases**

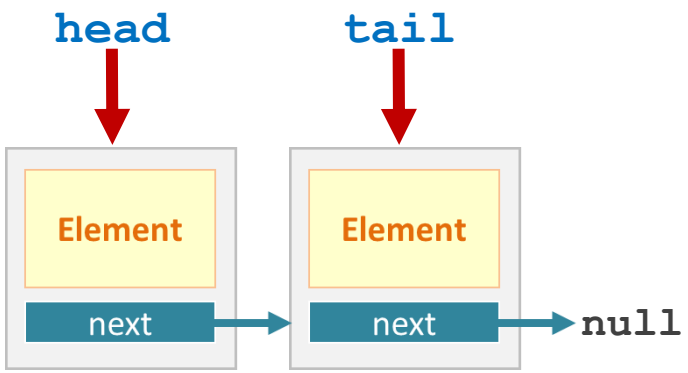
Empty List



One Element

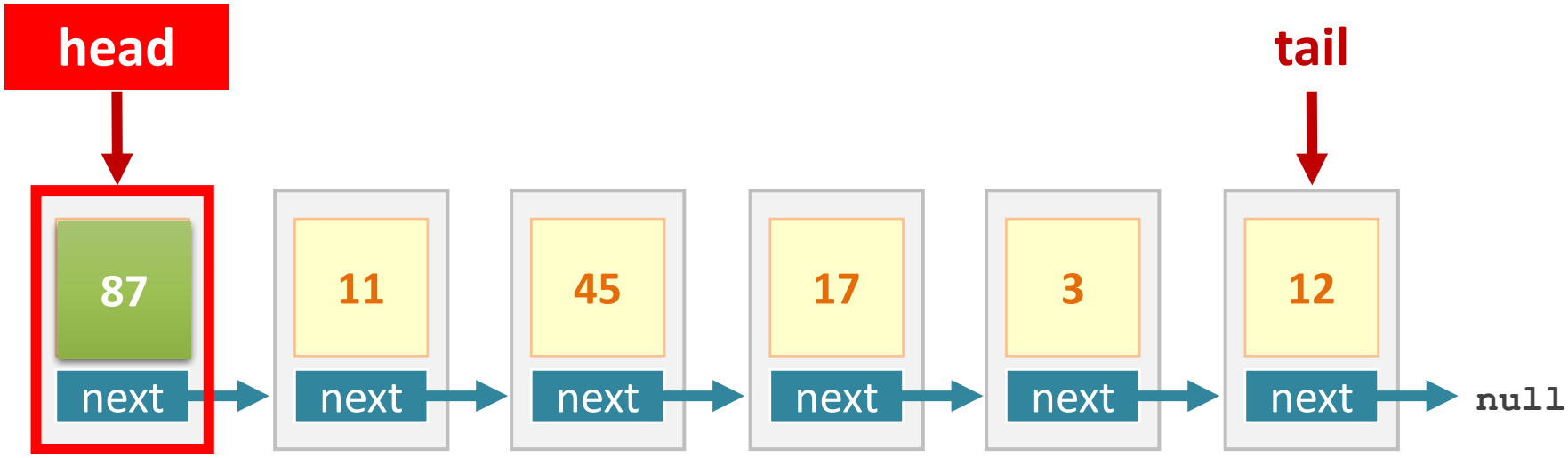


More Than One Element



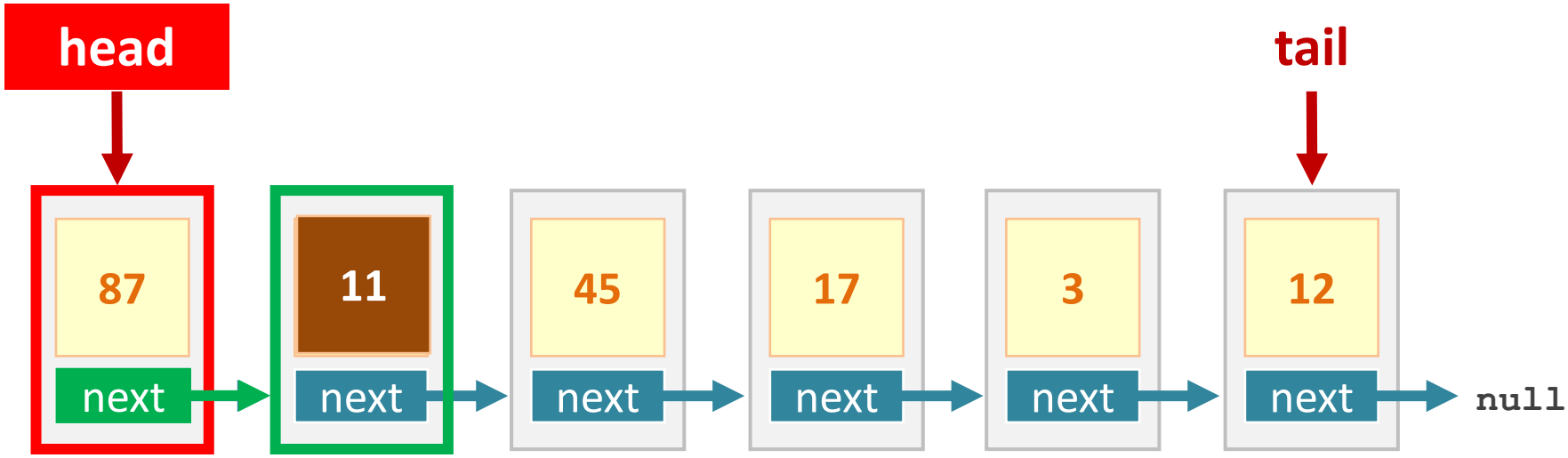
Practice: what is the value of?

What is the value of `head.element`?



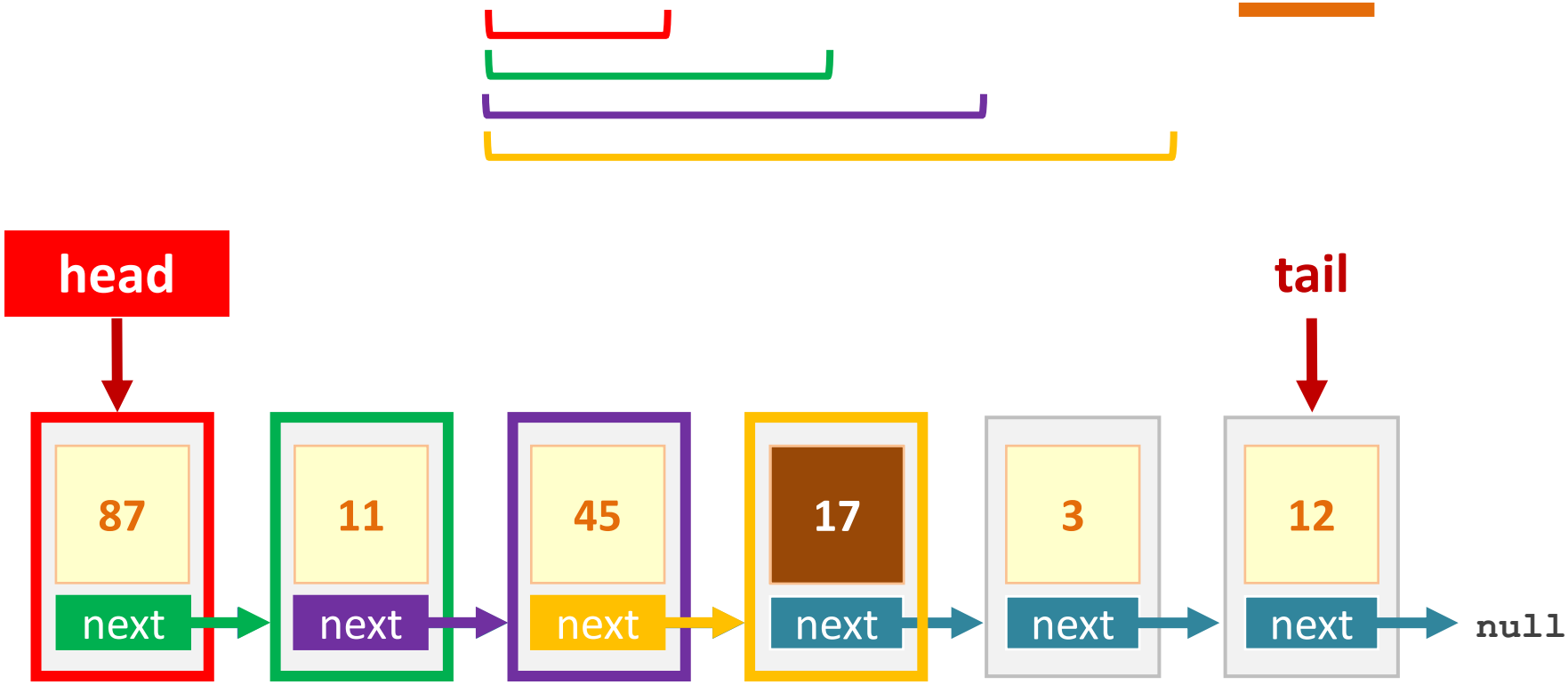
Practice: what is the value of?

What is the value of `head.next.element`?



Practice: what is the value of?

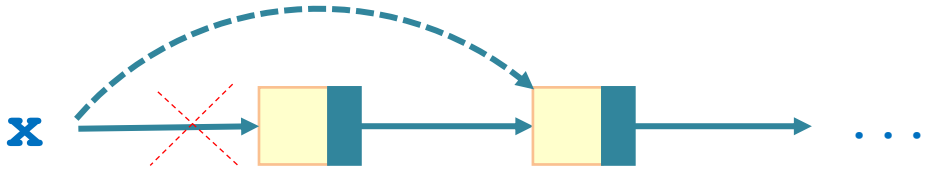
What is the value of `head.next.next.next.element`?



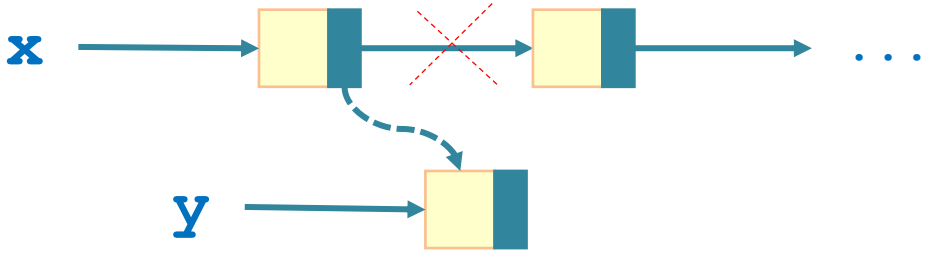
Useful illustrations…

Assume ***x and y*** are references to nodes in an linked list:

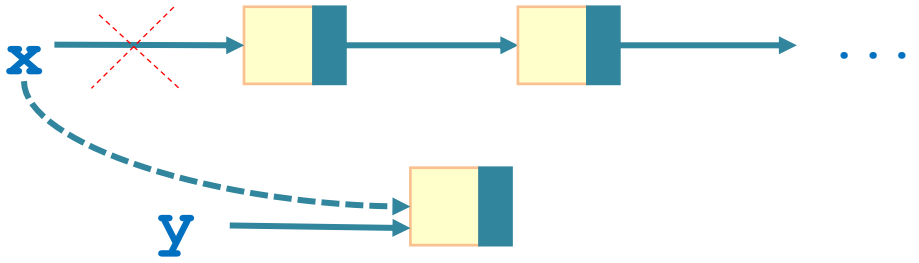
```
x = x.next
```




```
x.next = y
```

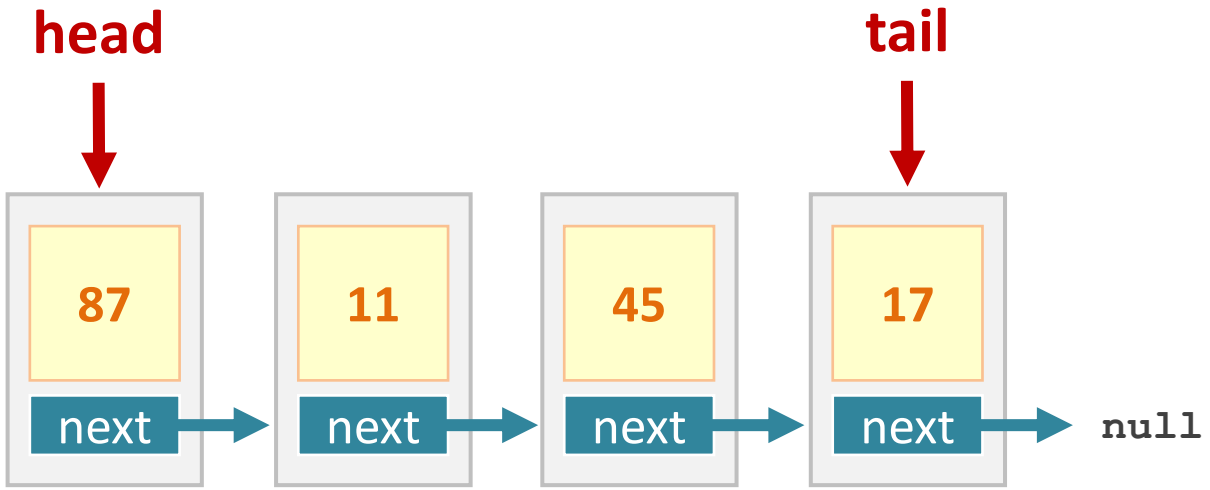



```
x = y
```



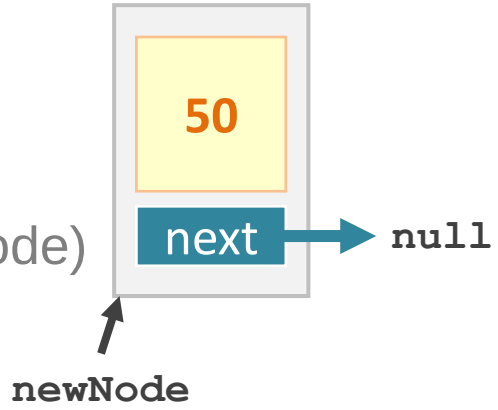
Practice: add an element to the end

Add an element = 50 to the end of this list



Algorithm:

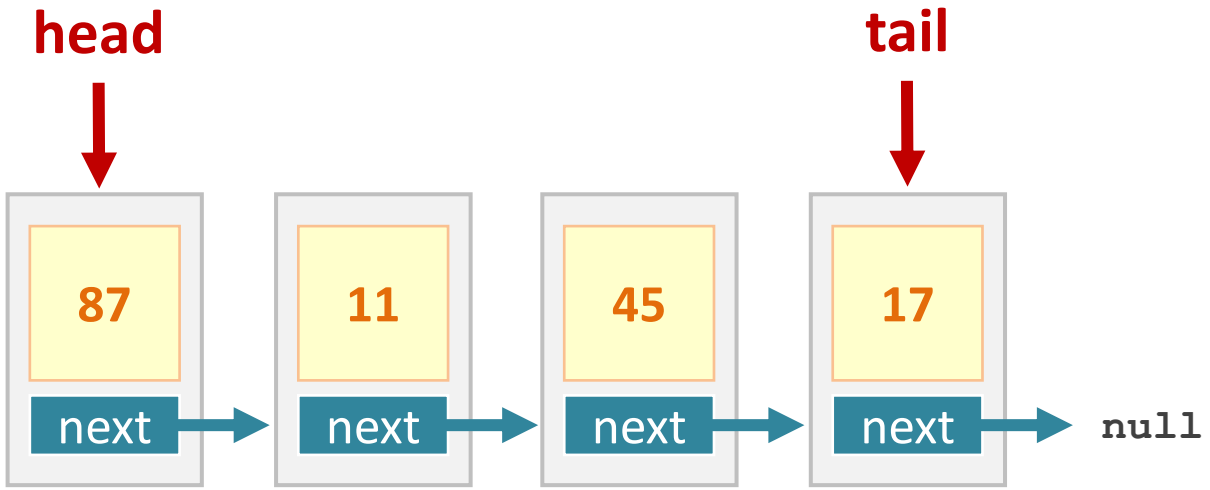
- 1) create newNode: element = 50, next = null
- 2) make tail.next refer to newNode (tail.next=newNode)
- 3) move tail to refer to newNode (tail=newNode)



Does the order matter??

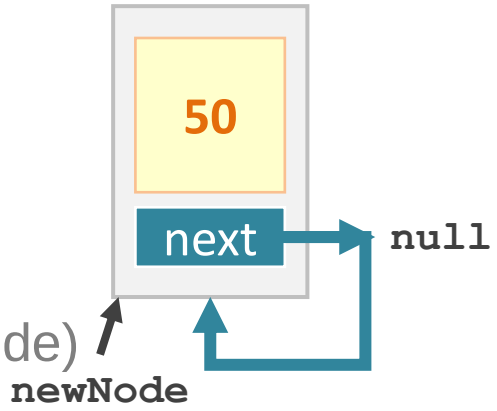
Let's try to change the order

Add an element = 50 to the end of this list



Algorithm:

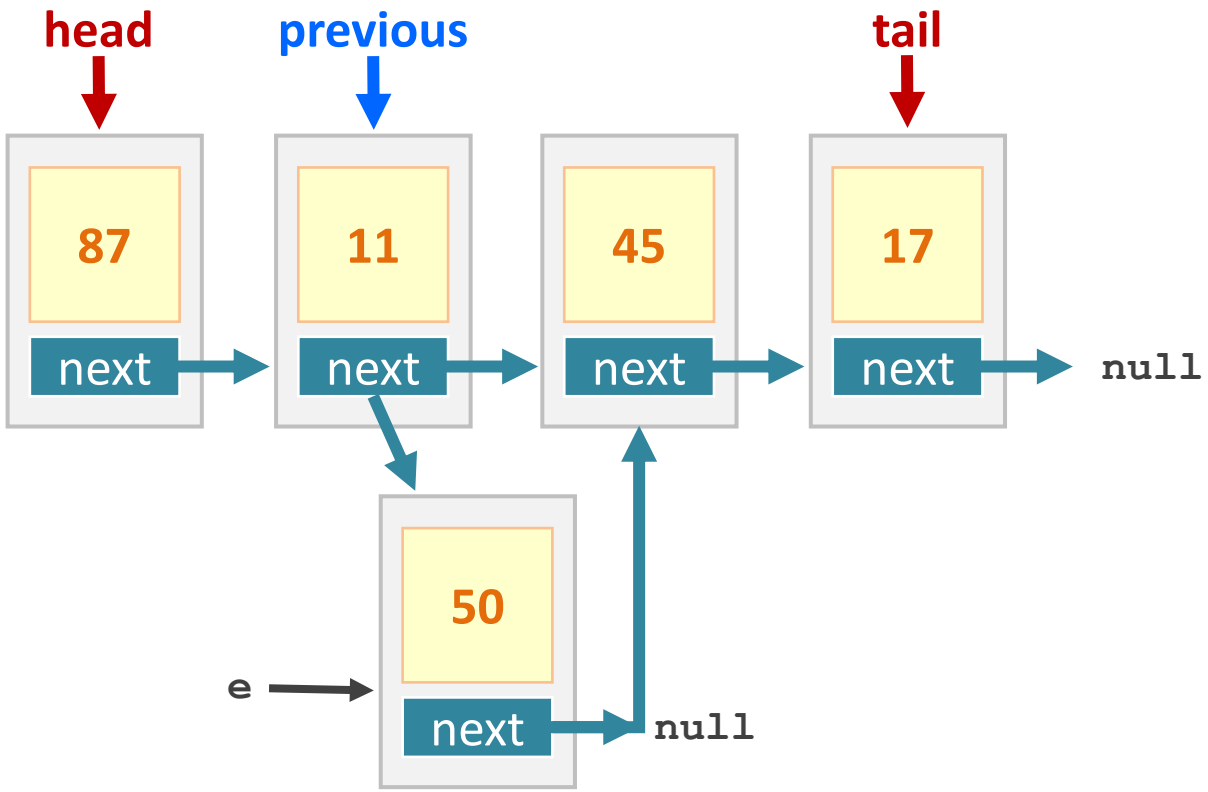
- 1) create newNode: element = 50, next = null
- 2) move tail to refer to newNode (tail=newNode)
- 3) make tail.next refer to newNode(tail.next=newNode)



Doesn't work!!!

Practice: insert an element

Insert a given node (element = 50, next = null) at index 2



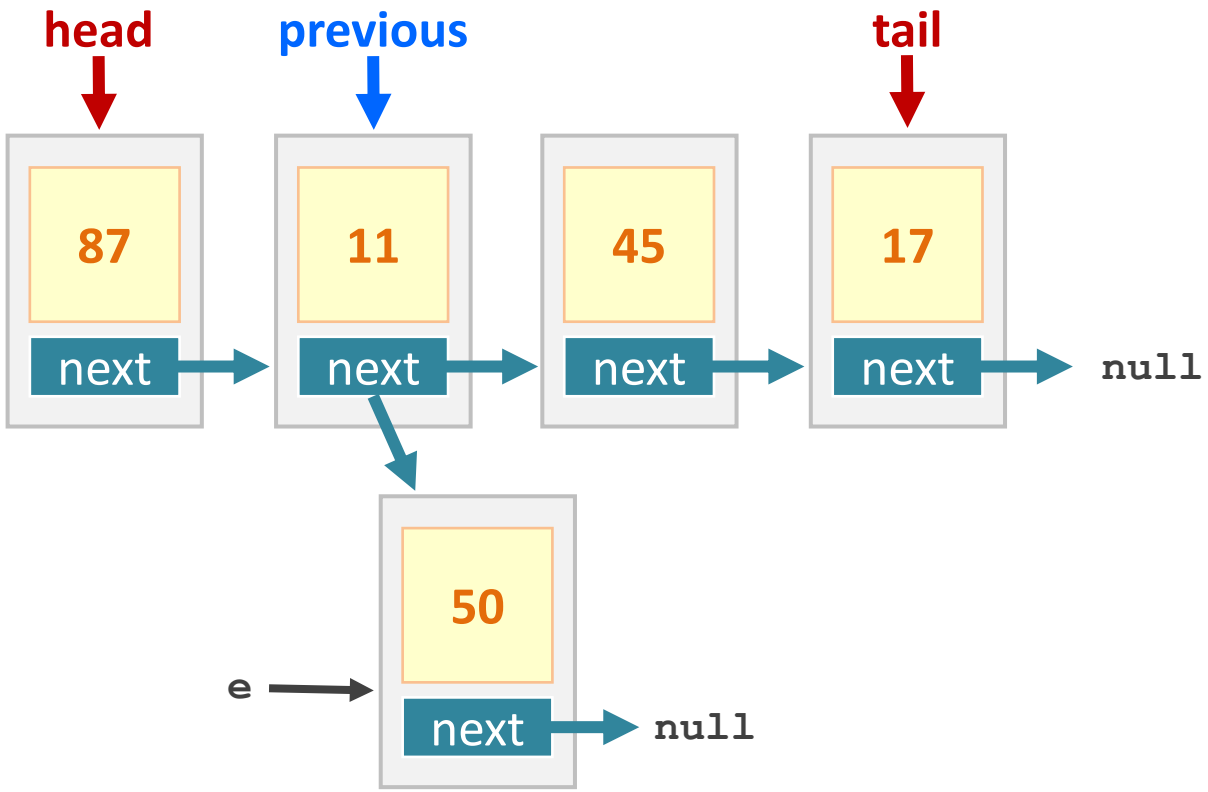
Algorithm:

- 1) get reference to the previous node, i.e. at index = 1
- 2) make `e.next` refer to `previous.next`
- 3) make `previous.next` refer to `e`

Does the order matter??

Let's try to change the order

Insert a given node (element = 50, next = null) at index 2



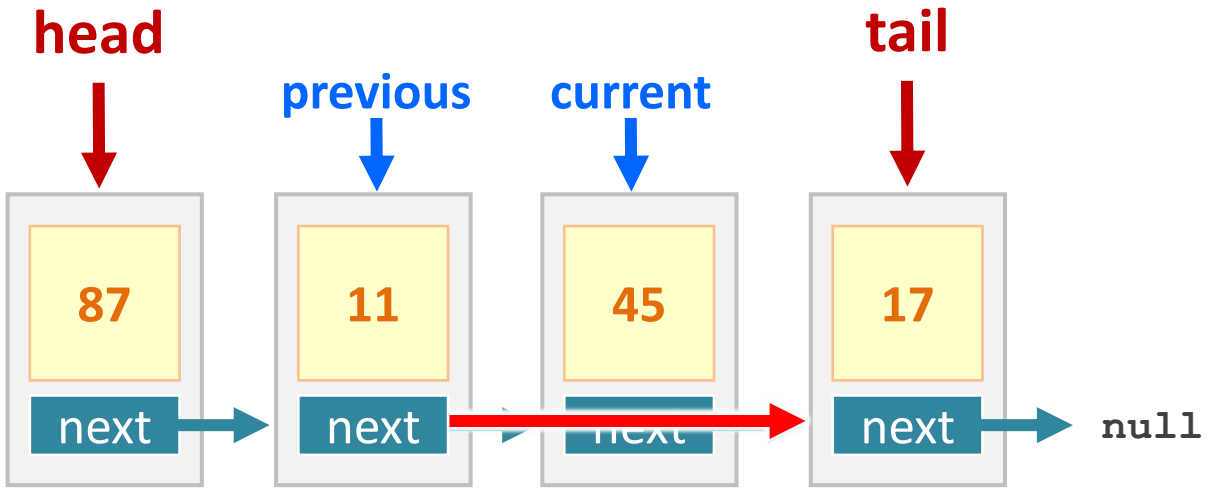
Algorithm:

- 1) get reference to the previous node, i.e. at index = 1
- 2) make `previous.next` refer to `e`
- Can't continue… node (45) is deleted
  - by the JVM garbage collector

Doesn't work!!!

Practice: Remove an element

Remove element at index = 2



Algorithm (we want ***11.next*** to point to ***17***):

- 1) get a reference to the required node and the previous one
- 2) make previous.next refer to current.next

Once current/previous references are deleted (when you get out of the remove method), the node is deleted.

Now… let's write some simple Java code…

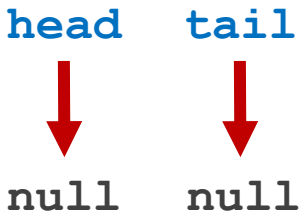
Remember:

- `next`, `head`, `tail` are of the type *Node*

Simple Practice 1: Add 3 nodes to a list

Write Java code to manually add 3 nodes to an empty linked list, and assume you have access to both head and tail

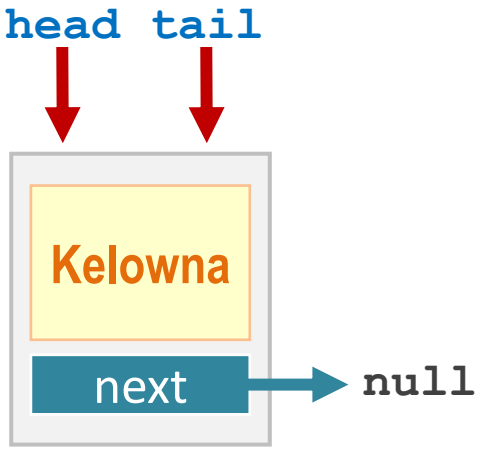
- Initially, *head = tail = null*



Solution:

- 1) Create a node and insert it to the list:

```
head = tail = new Node("Kelowna");
```

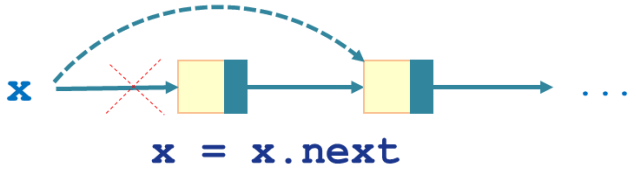


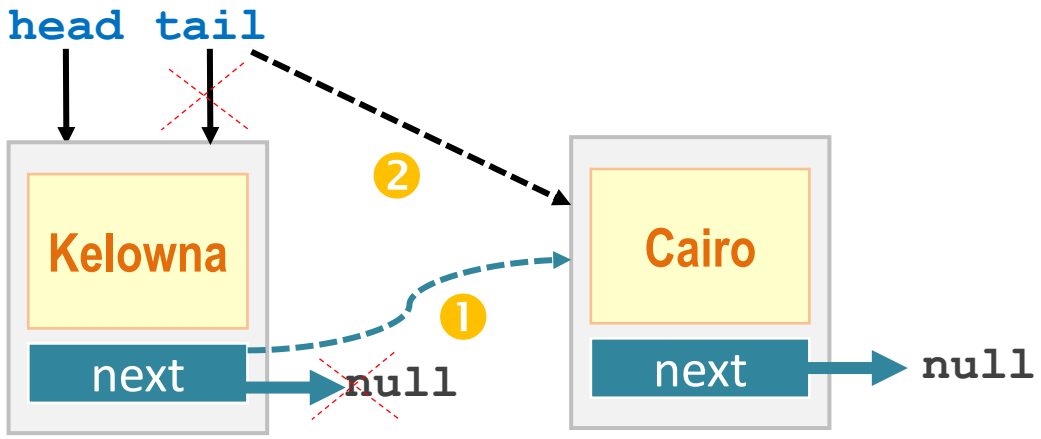
Simple Practice 1: Add 3 nodes to a list, cont'd

Solution, cont'd

- 2) Create the second node and add it to the list:

```
tail.next = new Node("Cairo");
tail = tail.next;
```



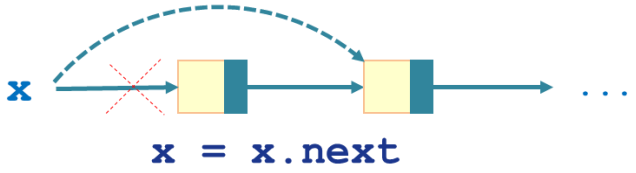


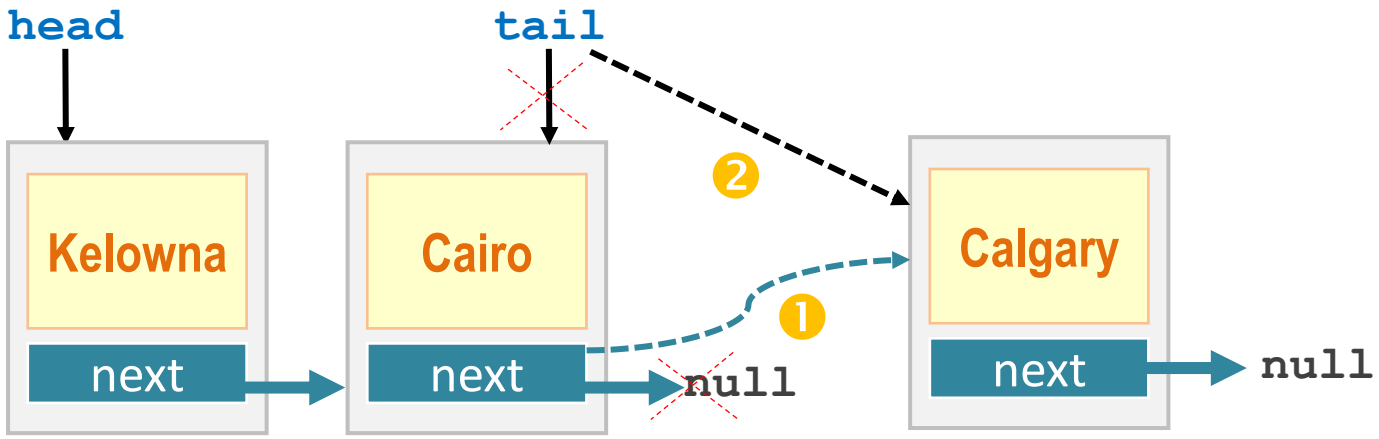
Simple Practice 1: Add 3 nodes to a list, cont'd

Solution, cont'd

- 3) Create the third node and add it to the list:

```
tail.next = new Node("Calgary");
tail = tail.next;
```





Simple Practice 2: Traversing All Elements in a List

Write Java code to visit all elements in a linkedlist (and do something. e.g. print the node's element)

Solution:

- Start from the beginning (head).
- Run a loop to visit each element and then progress to next element.
- When to stop the loop?
  - If the node is the last in the list, ***next*** will be null. You can use this property to detect the last node.

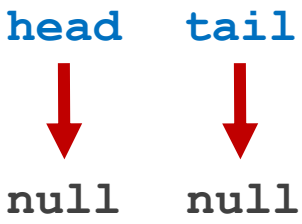
```
Node<E> current = head;
while (current != null) {
  //do something with current.element
  current = current.next;
}
```

Now… let's build a LinkedList !

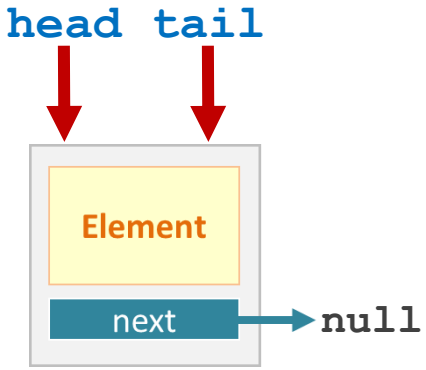
Remember:

- `next`, `head`, `tail` are of the type *Node*
- A node is deleted if it has no references pointing to it
- you have to consider all three cases shown below:

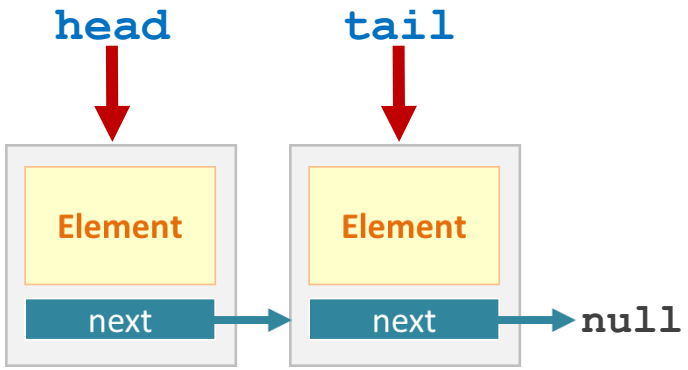
Empty List



One Element

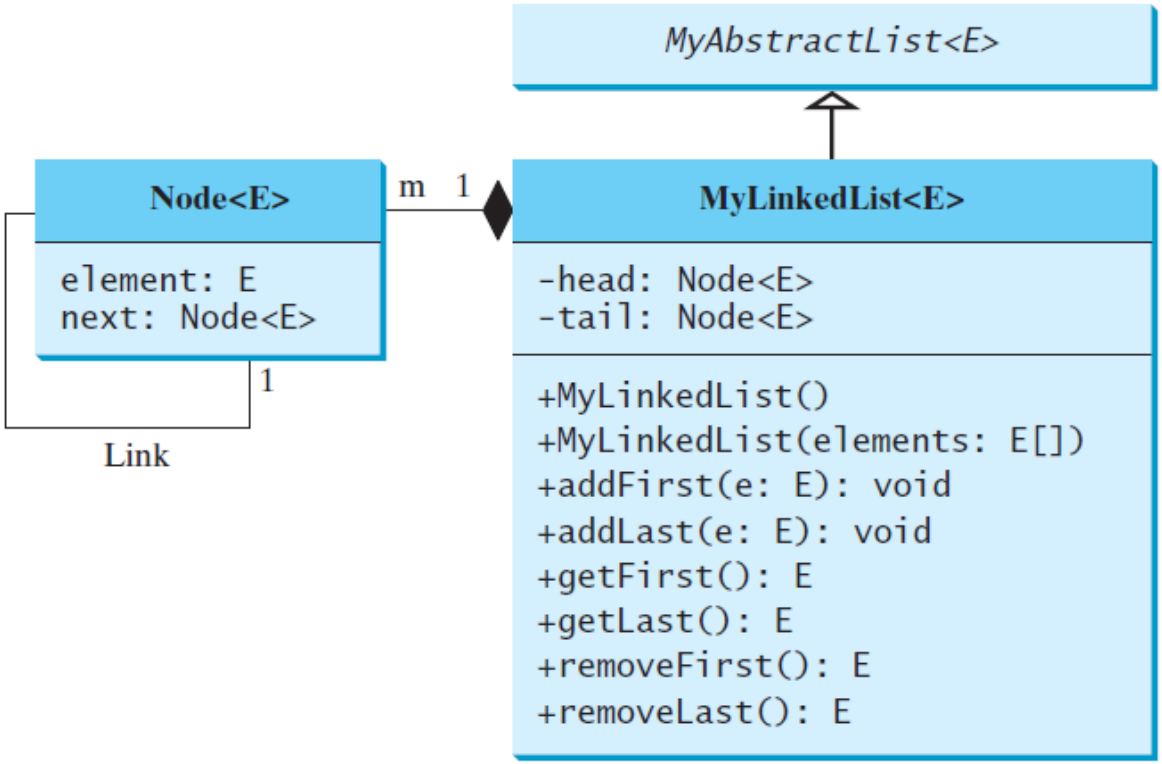


More Than One Element



Implementing `MyLinkedList`

In this part, we will implement the linked list shown below.



Creates a default linked list.
Creates a linked list from an array of elements.
Adds an element to the head of the list.
Adds an element to the tail of the list.
Returns the first element in the list.
Returns the last element in the list.
Removes the first element from the list.
Removes the last element from the list.

ADDING

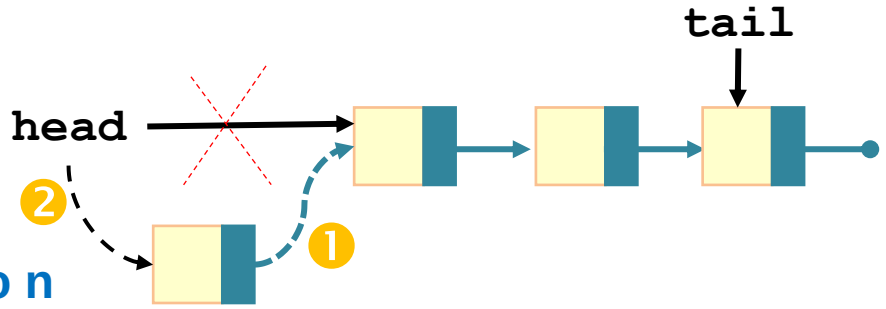
```
public void add(E e) {
    add(size, e);
}
```

We have already seen add(E e) in `MyAbstractList` class

Implement `addFirst(E e)` , `addLast(E e)` , `add(int i, E e)`

`addFirst(E e)`

- create a new node **n** to hold e
- **If empty list:**
  - **have both `head` and `tail` point to n**
- **else**
  - Have **n** point to first node
  - Have **head** point to **n**

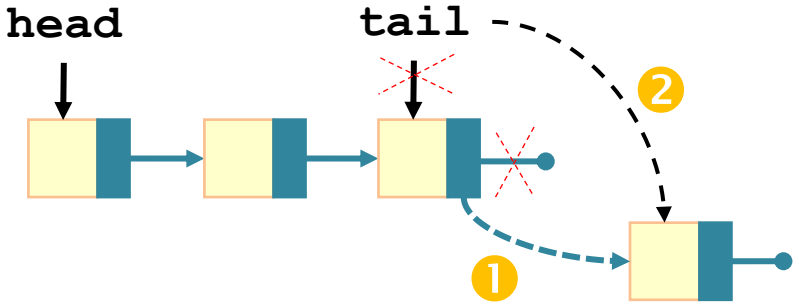



```
public void addFirst(E e) {
    Node<E> newNode = new Node<E>(e); // Create a new node
    if (tail == null)                 // if empty list
        head = tail = newNode;        // new node is the only node in list
    else{
        newNode.next = head;          // link the new node with the head
        head = newNode;               // head points to the new node
    }
    size++;
}
```

ADDING, cont.

addLast(E e)

- create a new node **n** to hold e
- **If empty list,**
  - **have both tail and head** point to n
- **else**
  - Have last node point to n
  - Have the tail point to n

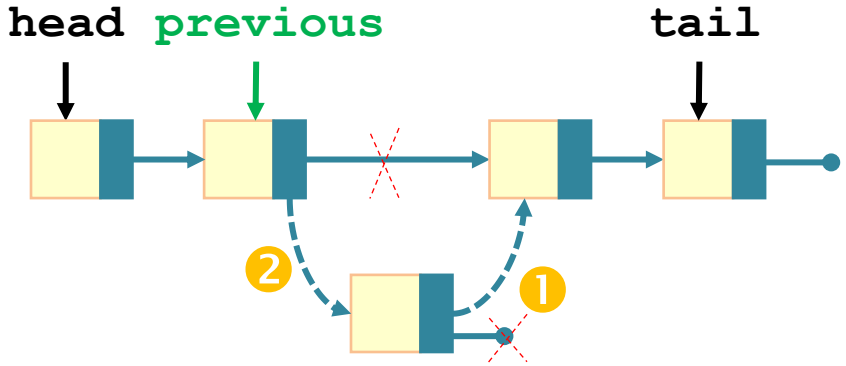



```
public void addLast(E e) {
    Node<E> newNode = new Node<E>(e); // Create a new for element e
    if (tail == null)                 // if empty list
        head = tail = newNode;        // new node is the only node in list
    else {
        tail.next = newNode;          // Link the new with the last node
        tail = tail.next;             // tail now points to the last node
    }
    size++;
}
```

ADDING, cont.

add(int index, E e)

- if invalid index → Exception
- else if index = 0 → addFirst
- else if index == size → addLast
- else
  - create a new node ***n*** to hold e
  - Get a reference ***previous*** to the element at index-1
  - Have the new node point to **previous.next** (i.e. n.next = previous.next)
  - Have **previous.next** point to **n**




```
public void add(int index, E e) {
    if(index <0 || index>size)
      throw new IndexOutOfBoundsException();    //according to Java doc
    else if(index == 0) addFirst(e);
    else if(index == size) addLast(e);
    else {
        Node<E> newNode = new Node<>(e);      // the new node
        Node<E> previous = head;              // ]
        for (int i = 0; i < index-1; i++)     // ]-get reference to idx-1
            previous = previous.next;         // ]
        newNode.next = previous.next;         // #1 in above diagram
        previous.next = newNode;              // #2 in above diagram
        size++;
    }
}
```