# Implementing Lists, Stacks, Queues, and Priority Queues

Paul Fodor

CSE260, Computer Science B: Honors

Stony Brook University

http://www.cs.stonybrook.edu/~cse260

# Objectives

- To **design common features of lists in an interface and** provide skeleton implementation in **an abstract class** for Collections.
- To design and implement a **dynamic list using an array**.
- To design and implement a **dynamic list using a linked structure**.
- To **design and implement a stack class** using an array list **and a queue class using a linked list**.
- To **design and implement a priority queue using a heap**.



(c) Paul Fodor (CS Stony Brook) & Pearson

# Lists

- A ***list*** is a popular data structure to store data in **sequential** order
  - For example, a list of customers, a list of books, a list of courses, a list of students, a list of cities, a list of available rooms, etc. can be stored using lists
  - The common operations on a list are usually the following:
    - ***Insert*** a new element to the list
    - ***Delete*** an element from the list
    - Find ***if an element is in the list***
    - ***Retrieve*** an element from the list
    - Find ***how many*** elements are in the list
    - Find if the list is ***empty***



(c) Paul Fodor (CS Stony Brook) & Pearson

# Two Ways to Implement Lists

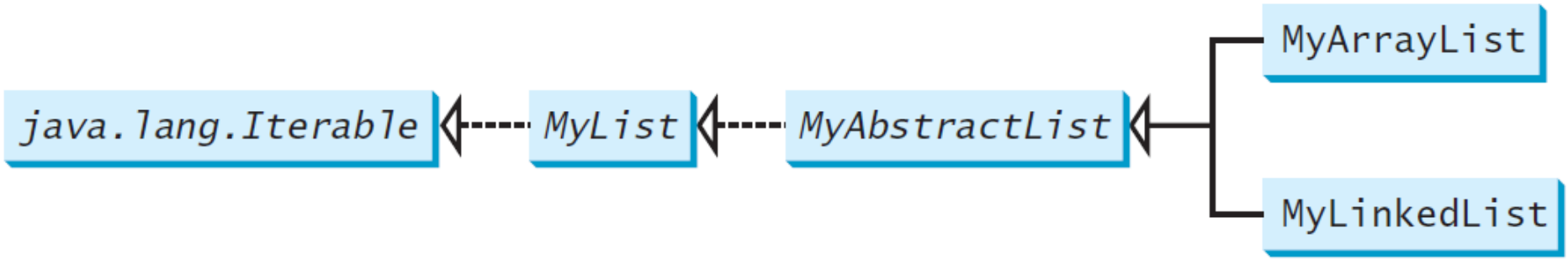
- There are two ways to implement a list:
  - ***Using arrays*** to store the elements: the array is dynamically expanded:
    - If the capacity of the array is exceeded, create a new larger array and copy all the elements from the current array to the new array
  - ***Using linked list*** consisting of nodes
    - Each node is dynamically created to hold an ***element***
    - All the nodes are linked together to form a list
- For convenience, let's name these two classes: **`MyArrayList`** and **`MyLinkedList`**



(c) Paul Fodor (CS Stony Brook) & Pearson

# Design of `ArrayList` and `LinkedList`

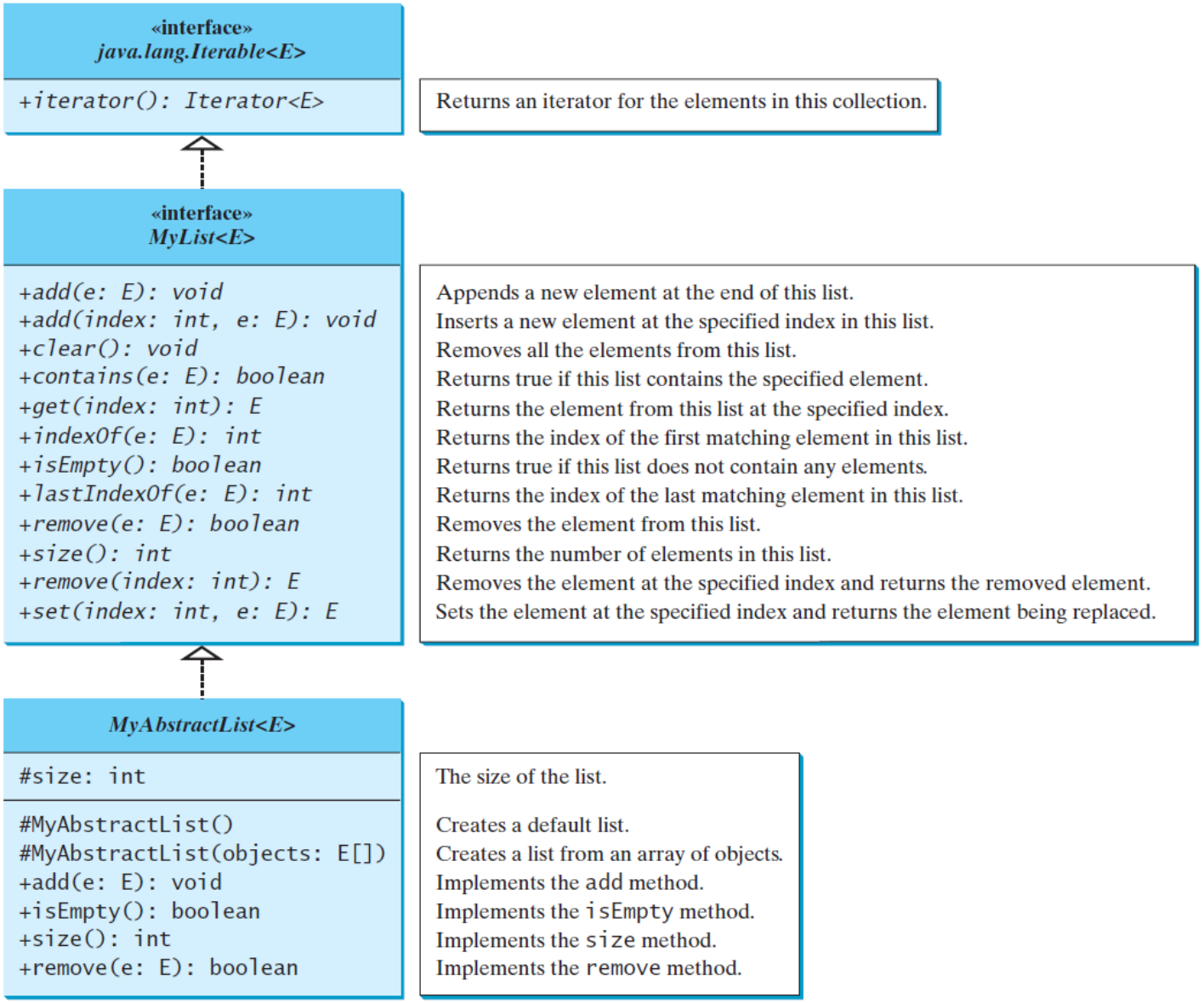
- The two classes have common operations, but different data fields
  - The common operations can be generalized in an interface or an abstract class
  - A good strategy is to combine the virtues of interfaces and abstract classes by providing both interface and abstract class in the design so the user can use either the interface or the abstract class whichever is convenient -> Such an abstract class is known as a ***convenience class***






(c) Paul Fodor (CS Stony Brook) & Pearson

# `MyList` Interface and `MyAbstractList` Class






(c) Paul Fodor (CS Stony Brook) & Pearson

```
public interface MyList<E> extends 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);
```


(c) Paul Fodor (CS Stony Brook) & Pearson

```
  /** 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 new set. */
  public E set(int index, E e);

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

}
```



(c) Paul Fodor (CS Stony Brook) & Pearson

```
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]);
  }

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

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

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



(c) Paul Fodor (CS Stony Brook) & Pearson

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

}
```



(c) Paul Fodor (CS Stony Brook) & Pearson

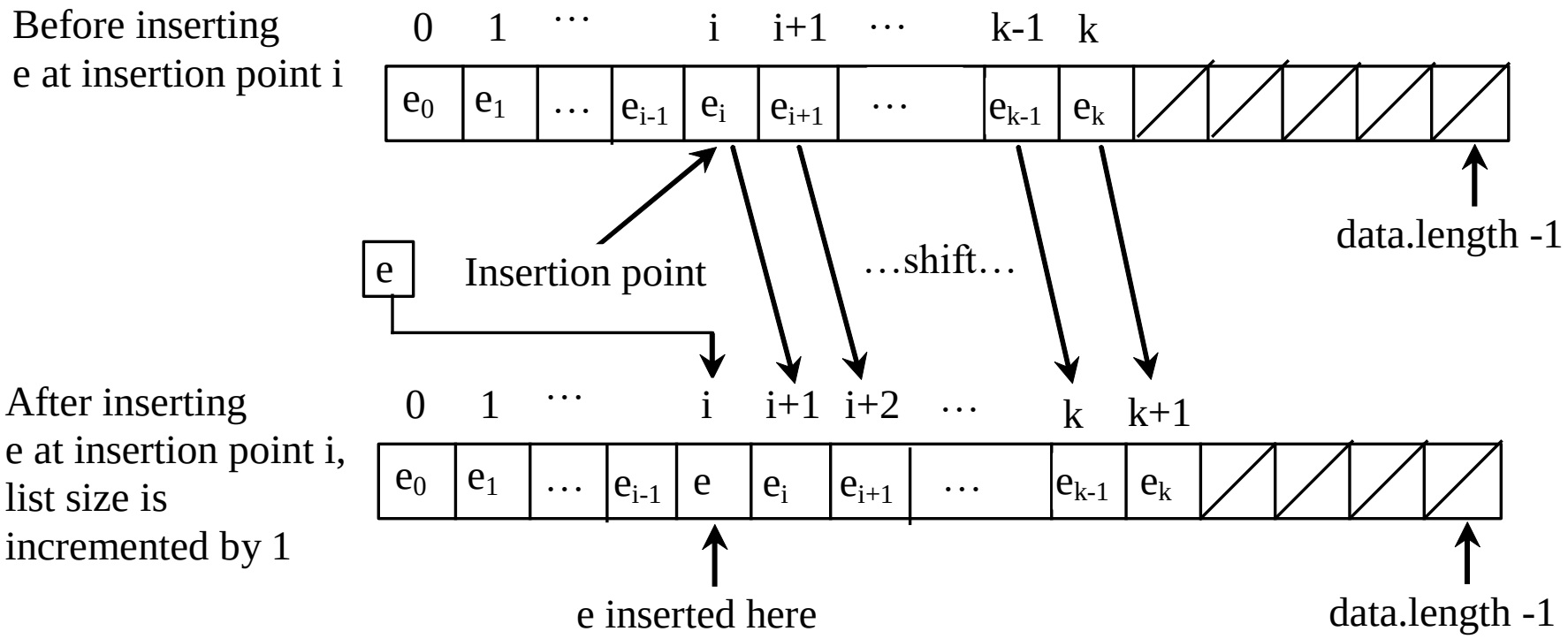
# Array Lists

- Array is a fixed-size data structure:
  - Once an array is created, its size cannot be changed
- However, we can still use array to implement dynamic data structures by creating 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 and the new array now becomes the current array


(c) Paul Fodor (CS Stony Brook) & Pearson

# 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

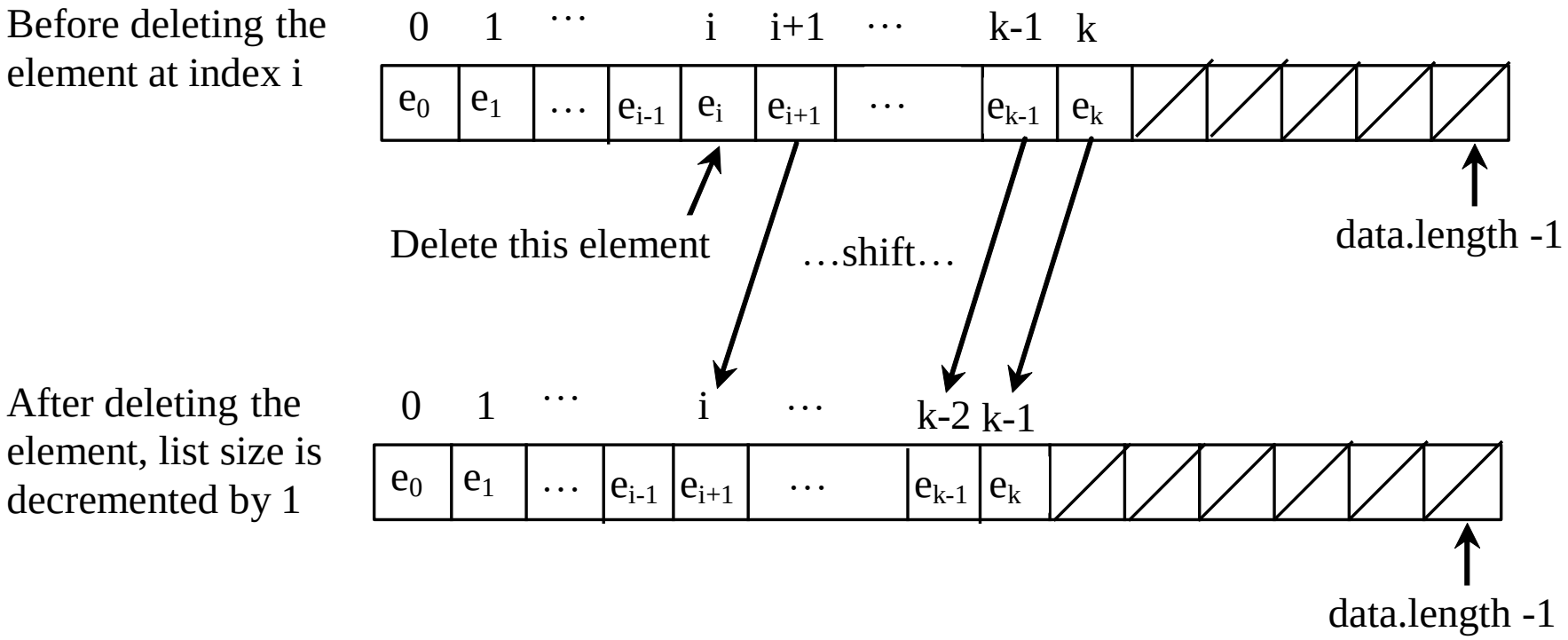





(c) Paul Fodor (CS Stony Brook) & Pearson

# Deletion

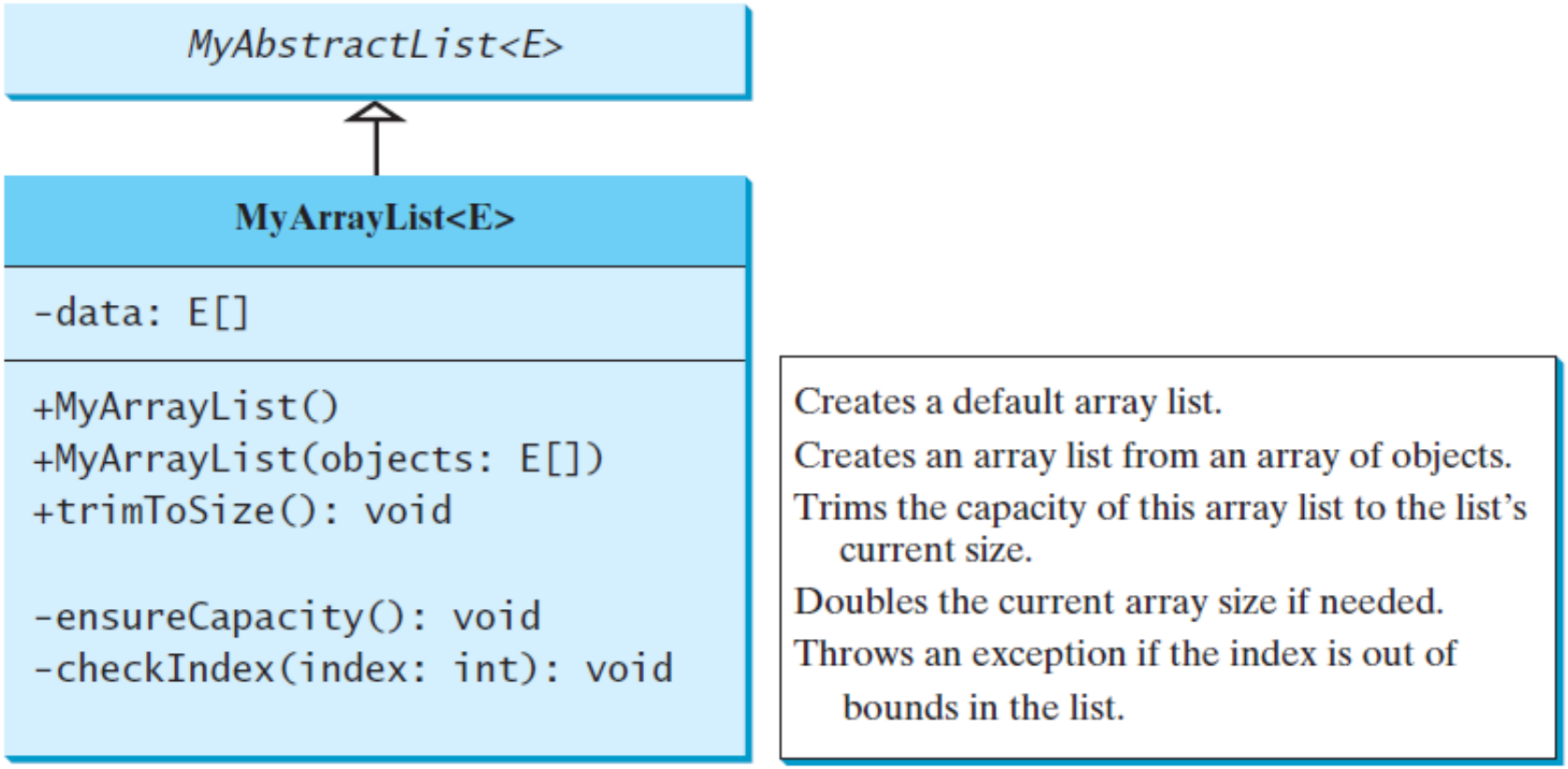
- To remove an element at a specified index, shift all the elements after the index to the left by one position and decrease the list size by 1






(c) Paul Fodor (CS Stony Brook) & Pearson

# Implementing MyArrayList






(c) Paul Fodor (CS Stony Brook) & Pearson

```
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) {
    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++;
  }
```



(c) Paul Fodor (CS Stony Brook) & Pearson

```
/** 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];
}
```



(c) Paul Fodor (CS Stony Brook) & Pearson

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

@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;
}
```



(c) Paul Fodor (CS Stony Brook) & Pearson

```
@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;
}

@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;
}
```



(c) Paul Fodor (CS Stony Brook) & Pearson

```
@Override
public String toString() {
  StringBuilder result = new StringBuilder(size*5).append("[");
  for (int i = 0; i < size; i++) {
    result.append(data[i]);
    if (i < size - 1) result.append(", ");
  }
  return result.append("]").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
}

@Override /** Override iterator() defined in Iterable */
public java.util.Iterator<E> iterator() {
  return new ArrayListIterator();
}
```



(c) Paul Fodor (CS Stony Brook) & Pearson

```
  private class ArrayListIterator
      implements java.util.Iterator<E> {
    private int current = 0; // Current index
    @Override
    public boolean hasNext() {
      return (current < size);
    }
    @Override
    public E next() {
      return data[current++];
    }
    @Override
    public void remove() {
      MyArrayList.this.remove(current);
    }
  }
}
```



(c) Paul Fodor (CS Stony Brook) & Pearson

```
public class TestMyArrayList {
  public static void main(String[] args) {
    // Create a list
    MyList<String> list = new MyArrayList<String>();

    // Add elements to the list
    list.add("America"); // Add it to the list
    System.out.println("(1) " + list);

    list.add(0, "Canada"); // Add it to the beginning of the list
    System.out.println("(2) " + list);

    list.add("Russia"); // Add it to the end of the list
    System.out.println("(3) " + list);

    list.add("France"); // Add it to the end of the list
    System.out.println("(4) " + list);

    list.add(2, "Germany"); // Add it to the list at index 2
    System.out.println("(5) " + list);

    list.add(5, "Norway"); // Add it to the list at index 5
    System.out.println("(6) " + list);
```



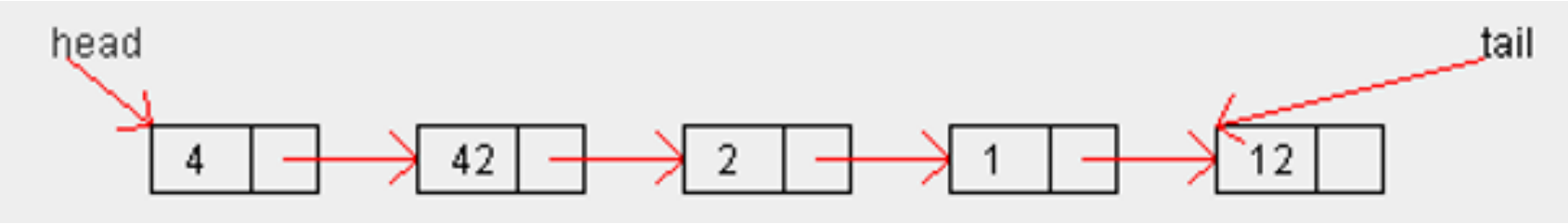
(c) Paul Fodor (CS Stony Brook) & Pearson

```
    // Remove elements from the list
    list.remove("Canada"); // Same as list.remove(0) in this case
    System.out.println("(7) " + list);

    list.remove(2); // Remove the element at index 2
    System.out.println("(8) " + list);

    list.remove(list.size() - 1); // Remove the last element
    System.out.print("(9) " + list + "\n(10) ");

    // enhanced-for loops require that the Collection implements
    //  Iterable or is an array
    for (String s: list)
      System.out.print(s.toUpperCase() + " ");
  }
}   Output:
    (1) [America]
    (2) [Canada, America]
    (3) [Canada, America, Russia]
    (4) [Canada, America, Russia, France]
    (5) [Canada, America, Germany, Russia, France]
    (6) [Canada, America, Germany, Russia, France, Norway]
    (7) [America, Germany, Russia, France, Norway]
    (8) [America, Germany, France, Norway]
    (9) [America, Germany, France]
    (10) AMERICA GERMANY FRANCE
```


(c) Paul Fodor (CS Stony Brook) & Pearson

# 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** (**if there is enough capacity**) are **very efficient**
- However, the methods **`addFirst(Object o)`** and **`removeFirst()`** are **inefficient** because they require shifting a large number of elements
  - ***Linked lists/structures*** improve the efficiency for adding and removing an element **at the beginning** of a list:

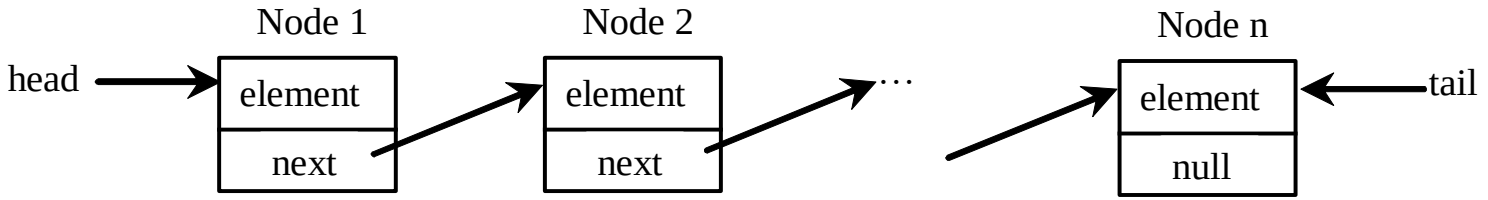





(c) Paul Fodor (CS Stony Brook) & Pearson

# Nodes in Linked Lists

- A linked list **consists of nodes:**
  - Each node contains an element/value, and each node is ***linked*** to its next neighbor:



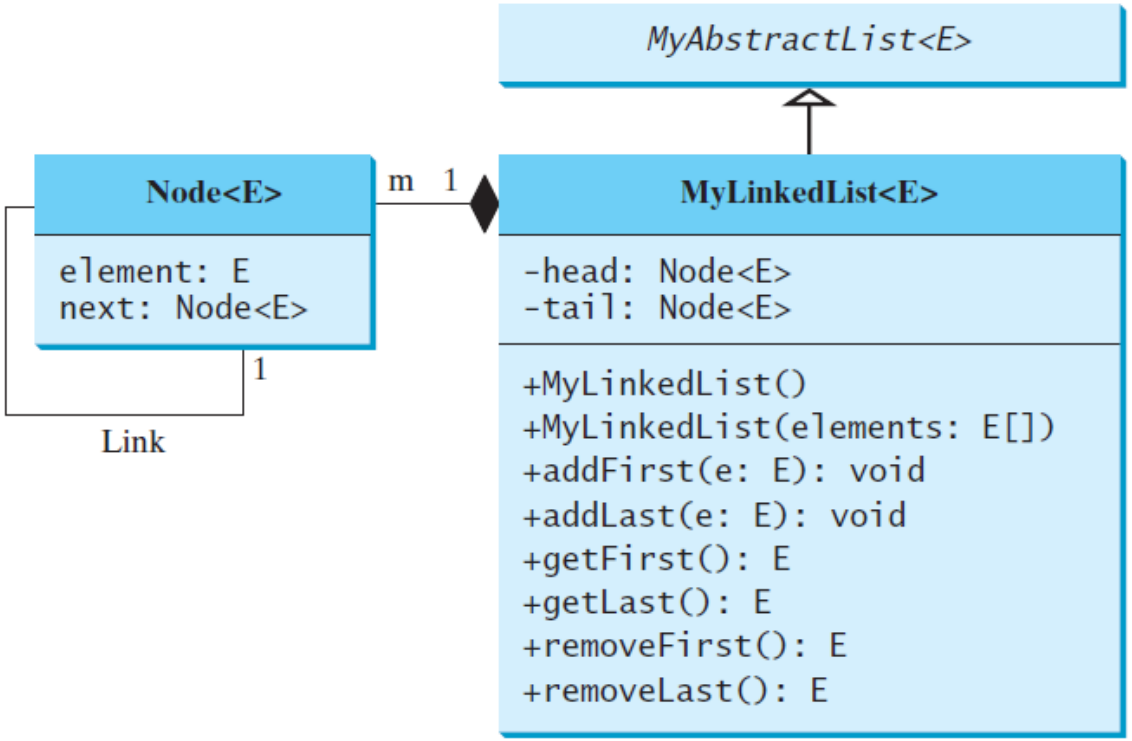

  - A node can be defined as a class, as follows:

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



(c) Paul Fodor (CS Stony Brook) & Pearson

# MyLinkedList



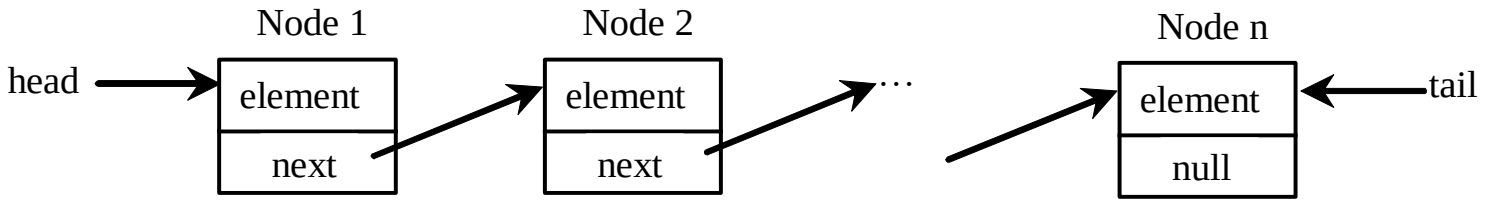

Creates a default linked list.
Creates a linked list from an array of elements.
Adds an element to the head of the list.
Adds an element to the tail of the list.
Returns the first element in the list.
Returns the last element in the list.
Removes the first element from the list.
Removes the last element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



(c) Paul Fodor (CS Stony Brook) & Pearson

# Empty list

- A data field variable **`head`** refers to the first node in the list, and a data field variable **`tail`** refers to the last node in the list




- If the list is empty, both are **`null`**:

Step 1: Declare head and tail:

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

The list is empty now



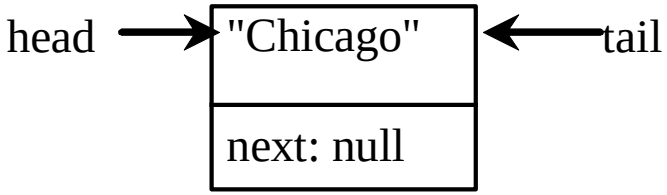
(c) Paul Fodor (CS Stony Brook) & Pearson

# Adding Nodes

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

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

After the first node is inserted



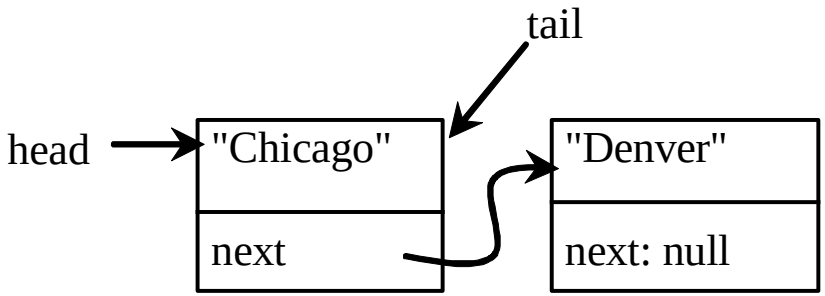



(c) Paul Fodor (CS Stony Brook) & Pearson

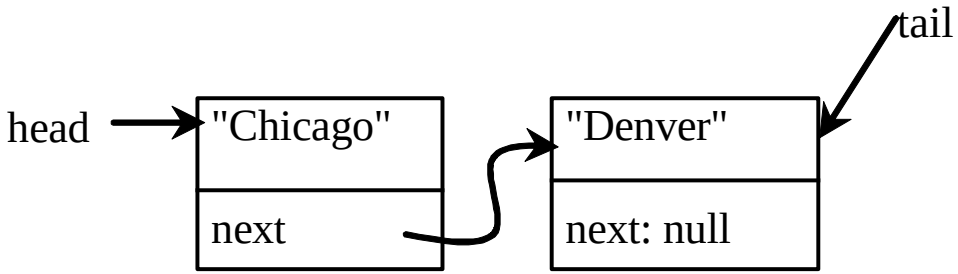
# Adding Nodes

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

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




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



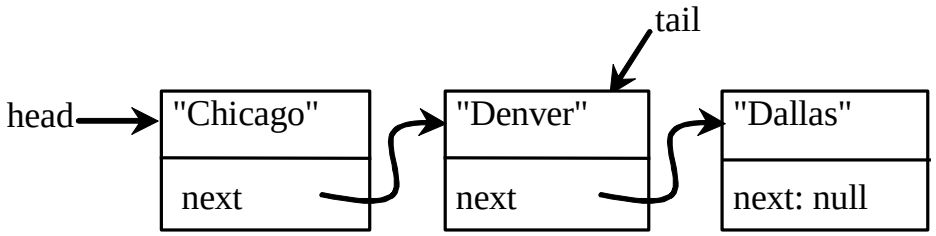



(c) Paul Fodor (CS Stony Brook) & Pearson

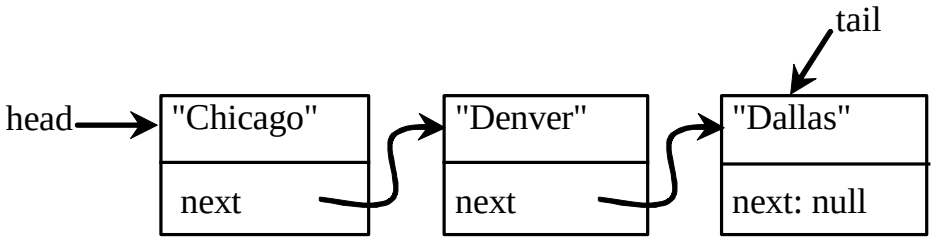
# Adding More Nodes is the same

Create the third node and insert it to the list:

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



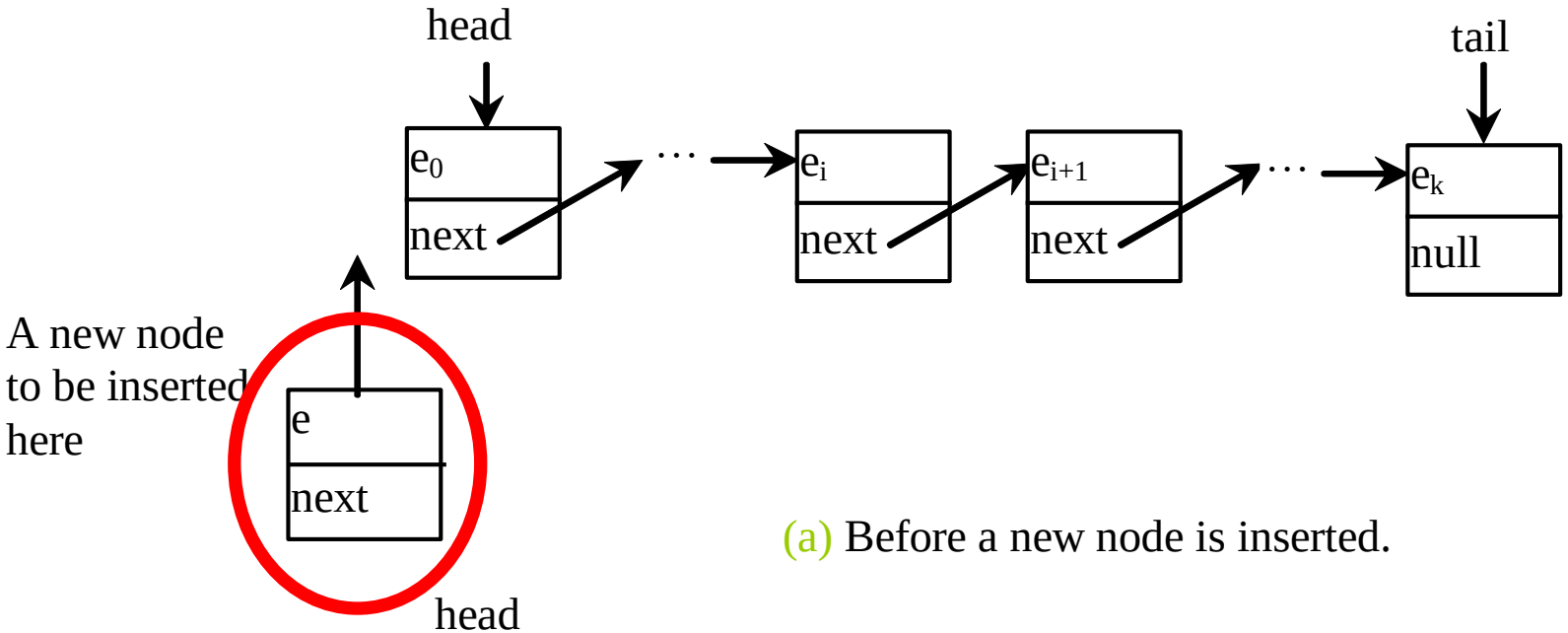

```
tail = tail.next;
```



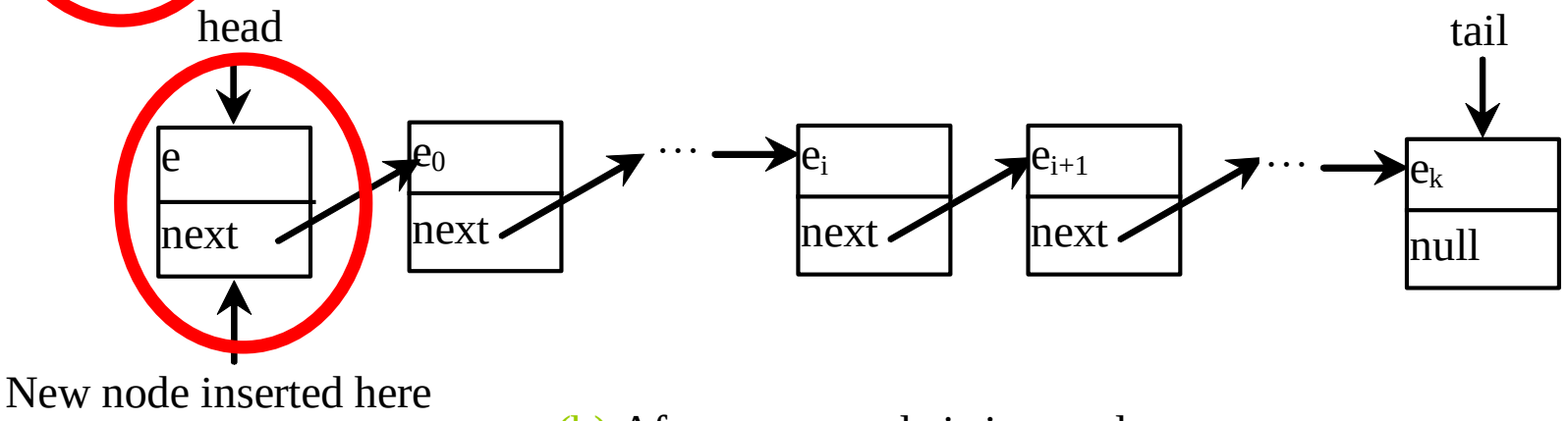



(c) Paul Fodor (CS Stony Brook) & Pearson

# Implementing `addFirst(E e)`




(a) Before a new node is inserted.

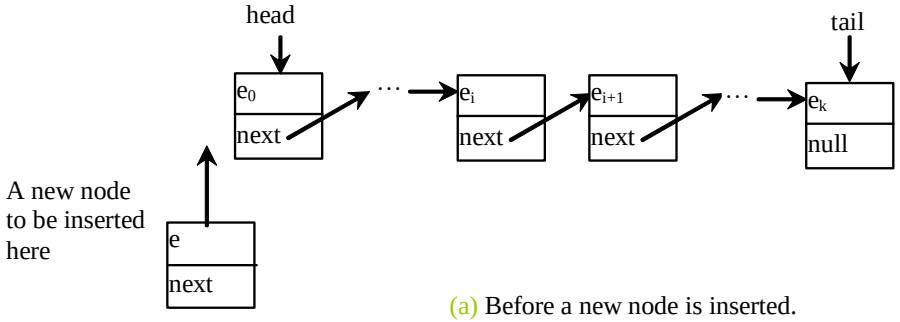



(b) After a new node is inserted.

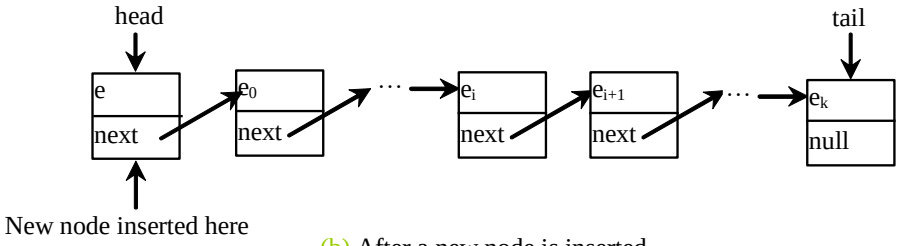


(c) Paul Fodor (CS Stony Brook) & Pearson

# Implementing `addFirst(E e)`




(a) Before a new node is inserted.



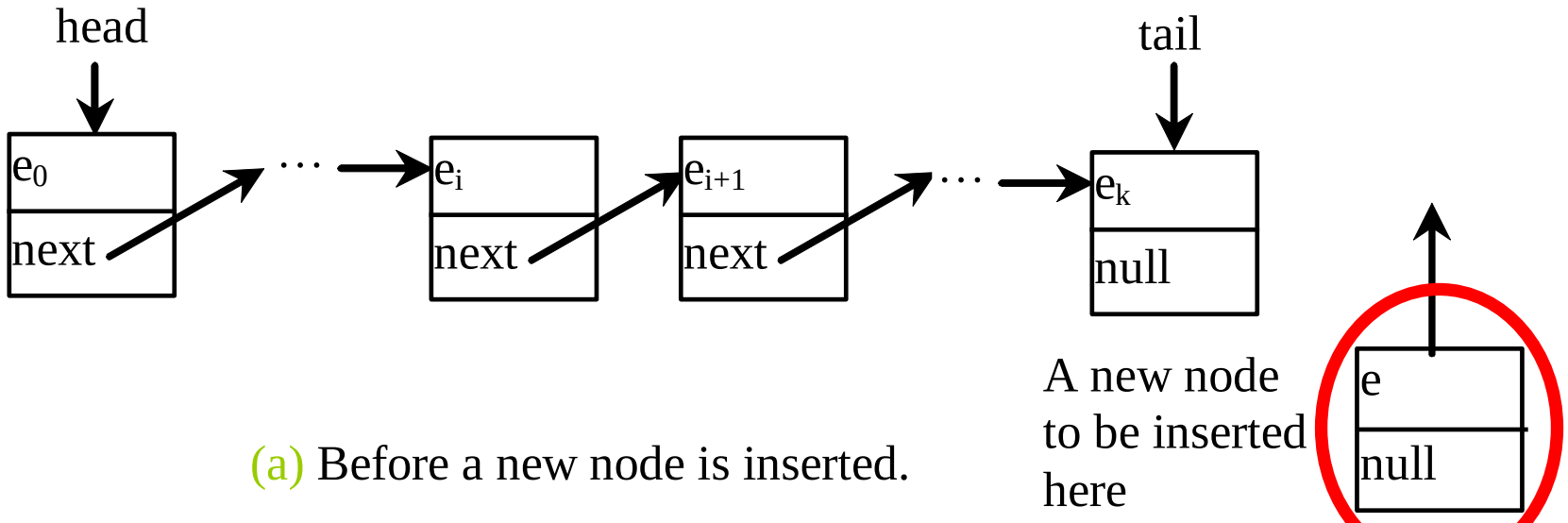

(b) After a new node is inserted.

```
public void addFirst(E e) {
  Node<E> newNode = new Node<>(e);
  newNode.next = head;
  head = newNode;
  size++;
  if (tail == null) // the list was empty before
    tail = head;
}
```

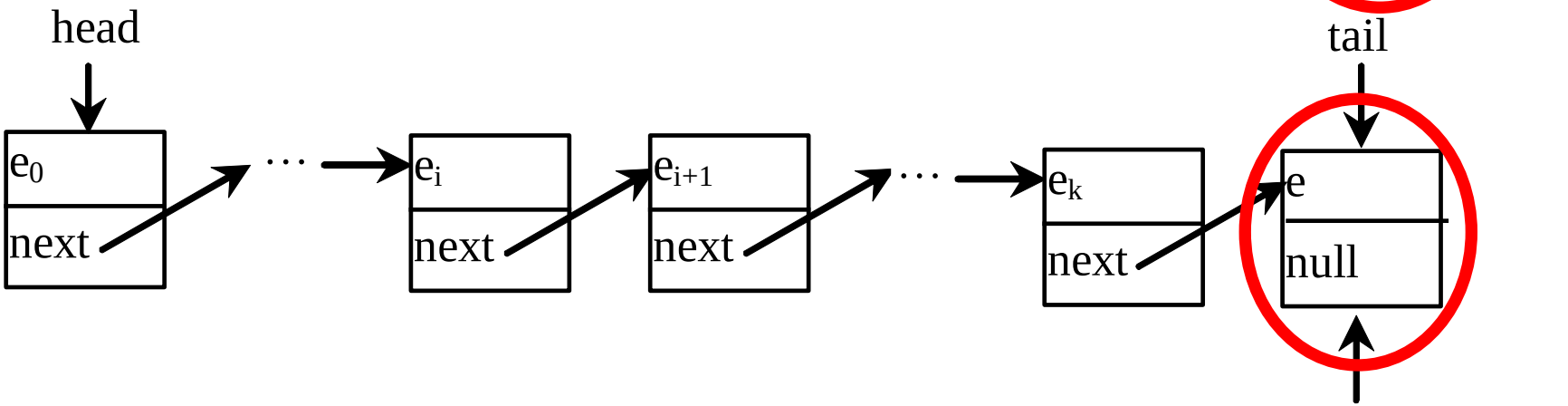


(c) Paul Fodor (CS Stony Brook) & Pearson

# Implementing `addLast(E e)`




(a) Before a new node is inserted.

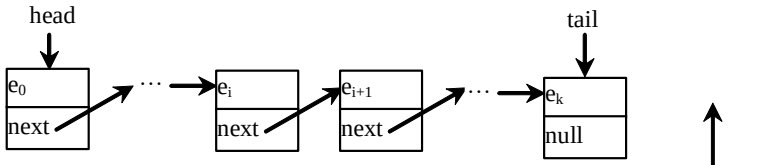



(b) After a new node is inserted.

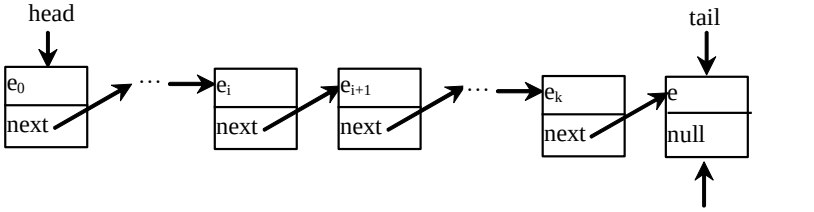


(c) Paul Fodor (CS Stony Brook) & Pearson

# Implementing `addLast(E e)`




(a) Before a new node is inserted.



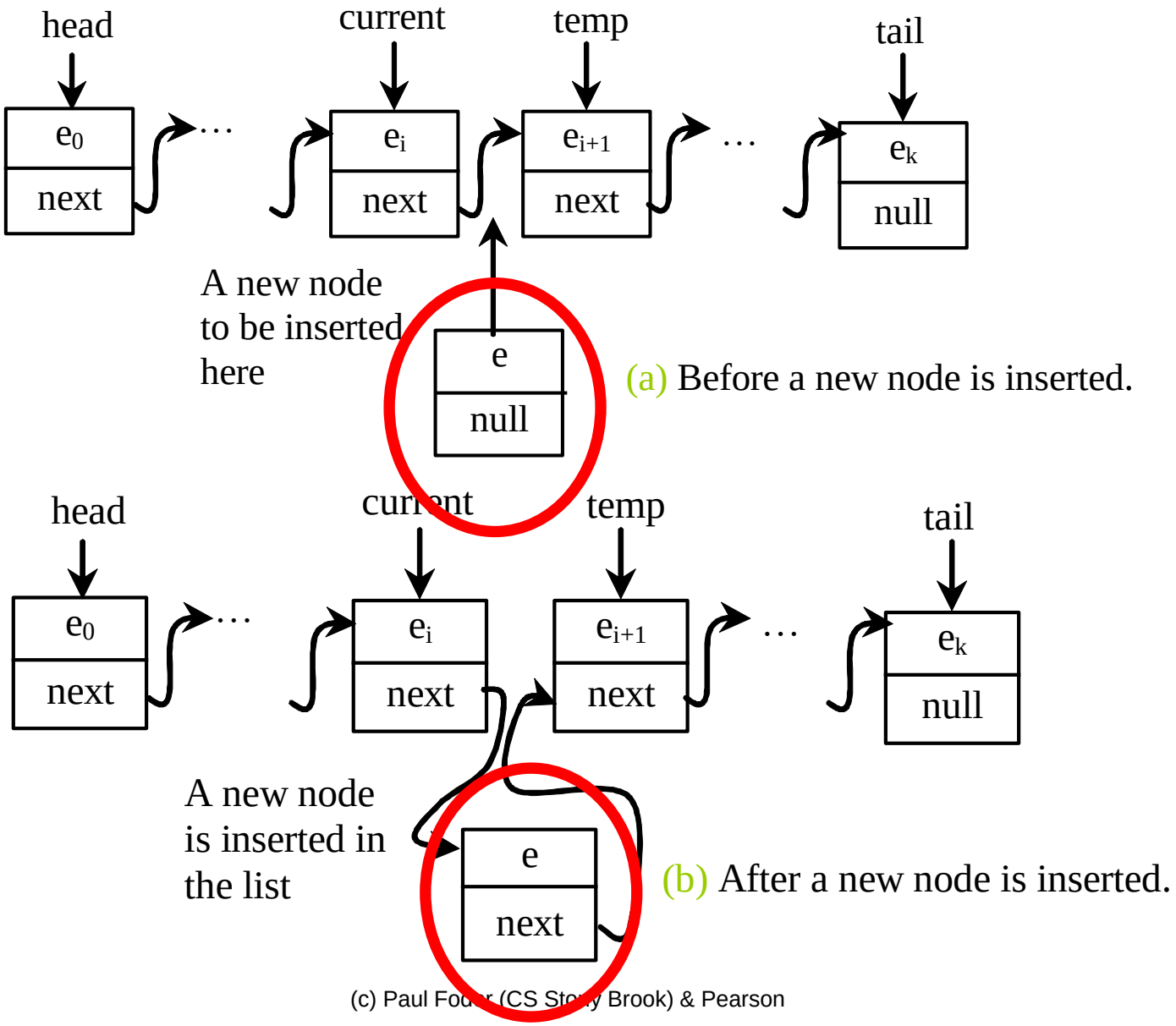

(b) After a new node is inserted.

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



(c) Paul Fodor (CS Stony Brook) & Pearson

# Implementing `add(int index, E e)`






(c) Paul Fodor (CS Stony Brook) & Pearson

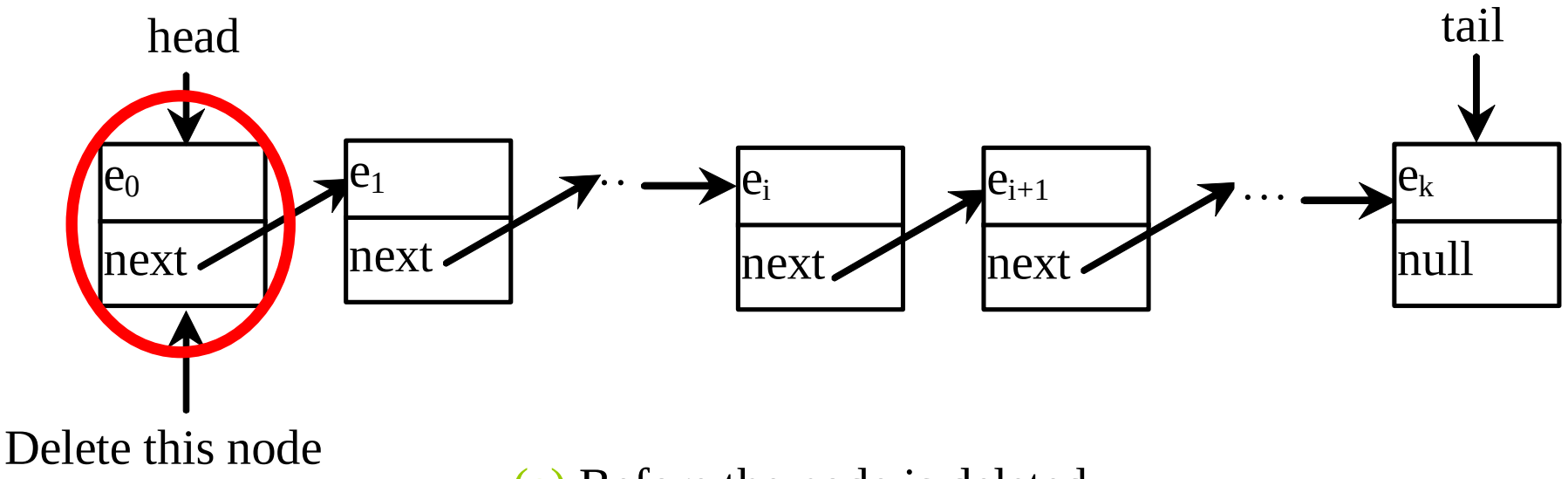
# Implementing `add(int index, E e)`

```
public void add(int index, E e) {
  if (index == 0) addFirst(e);
  else if (index == size) addLast(e);
  else if(index<0 || index>size)
    throw new RuntimeException();
  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);
    (current.next).next = temp;
    size++;
  }}
```

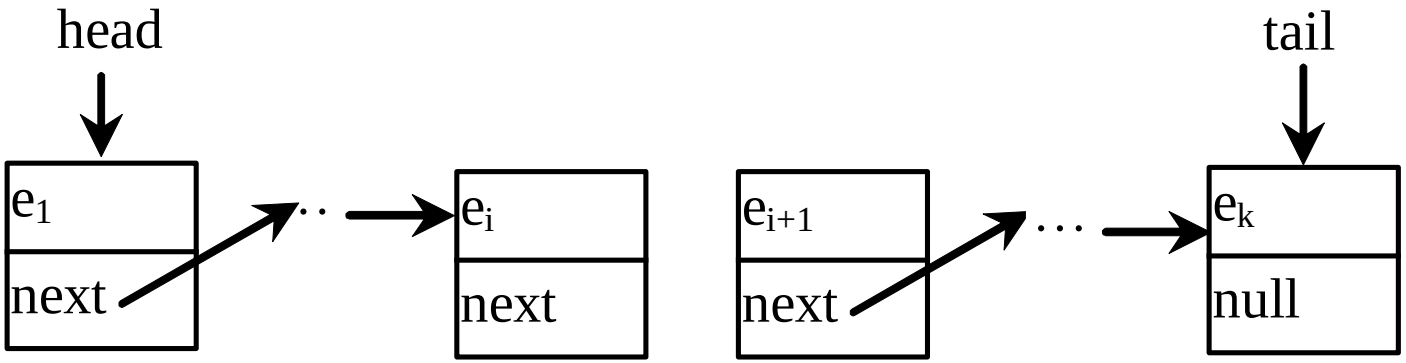


(c) Paul Fodor (CS Stony Brook) & Pearson

# Implementing `removeFirst()`

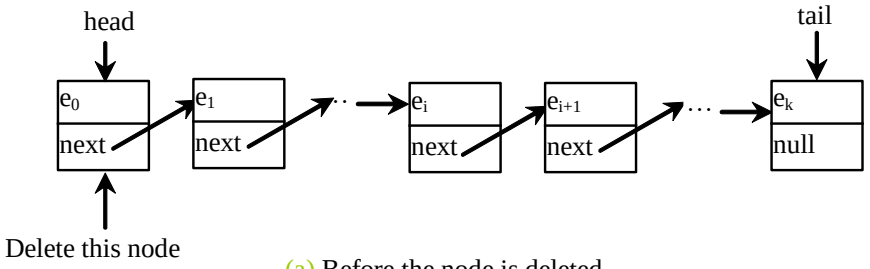



(a) Before the node is deleted.

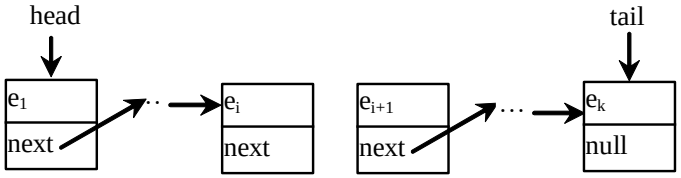



(b) After the first node is deleted

# Implementing `removeFirst()`




(a) Before the node is deleted.



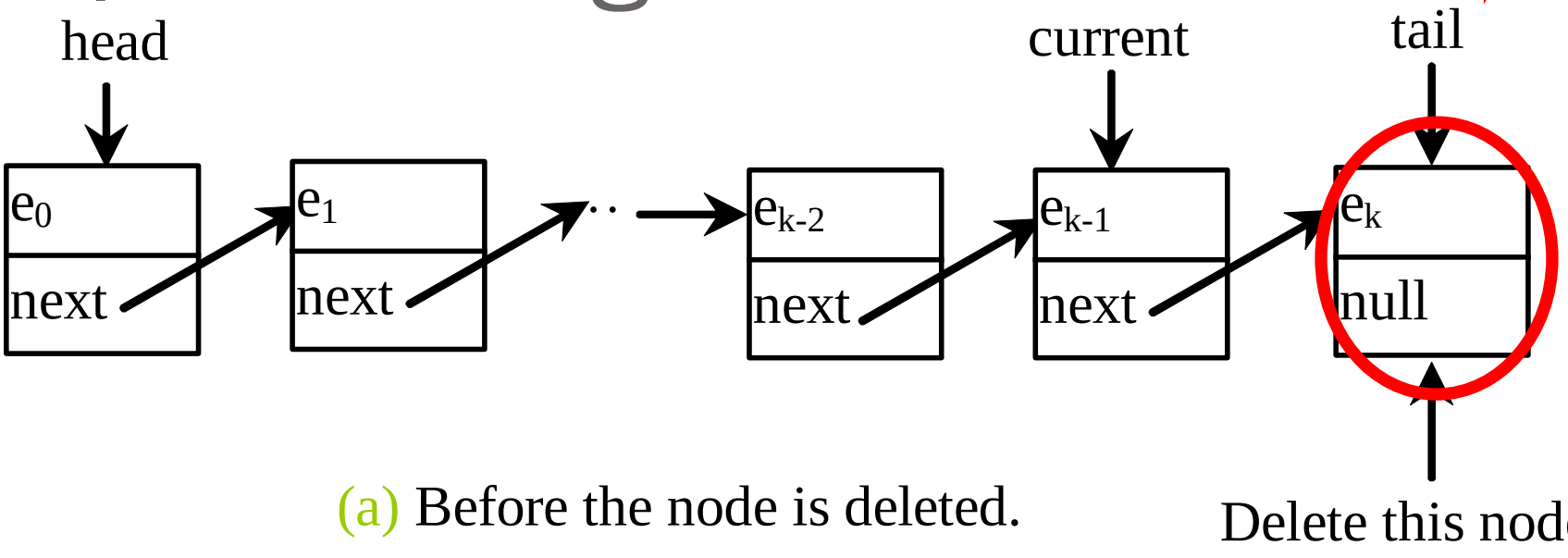

(b) After the first node is deleted

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

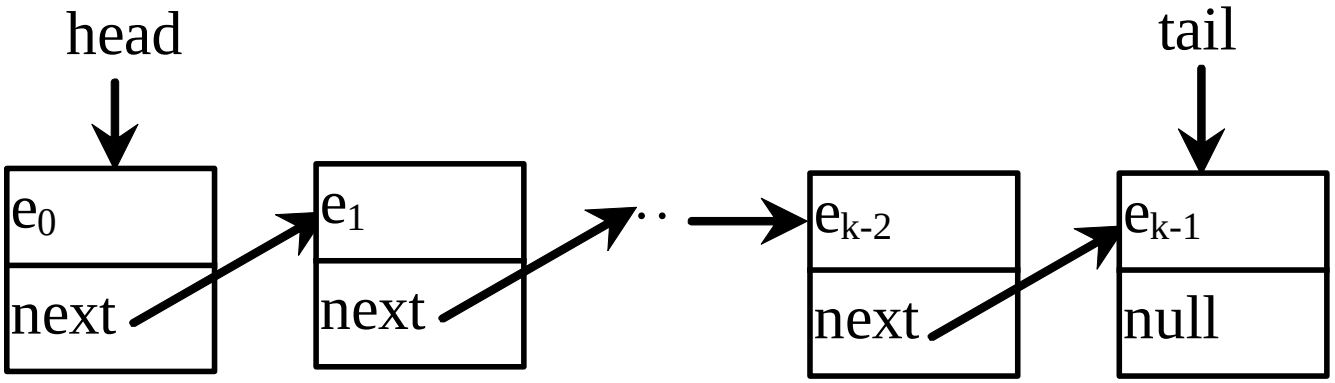


(c) Paul Fodor (CS Stony Brook) & Pearson

# Implementing `removeLast()`




(a) Before the node is deleted.




(b) After the last node is deleted

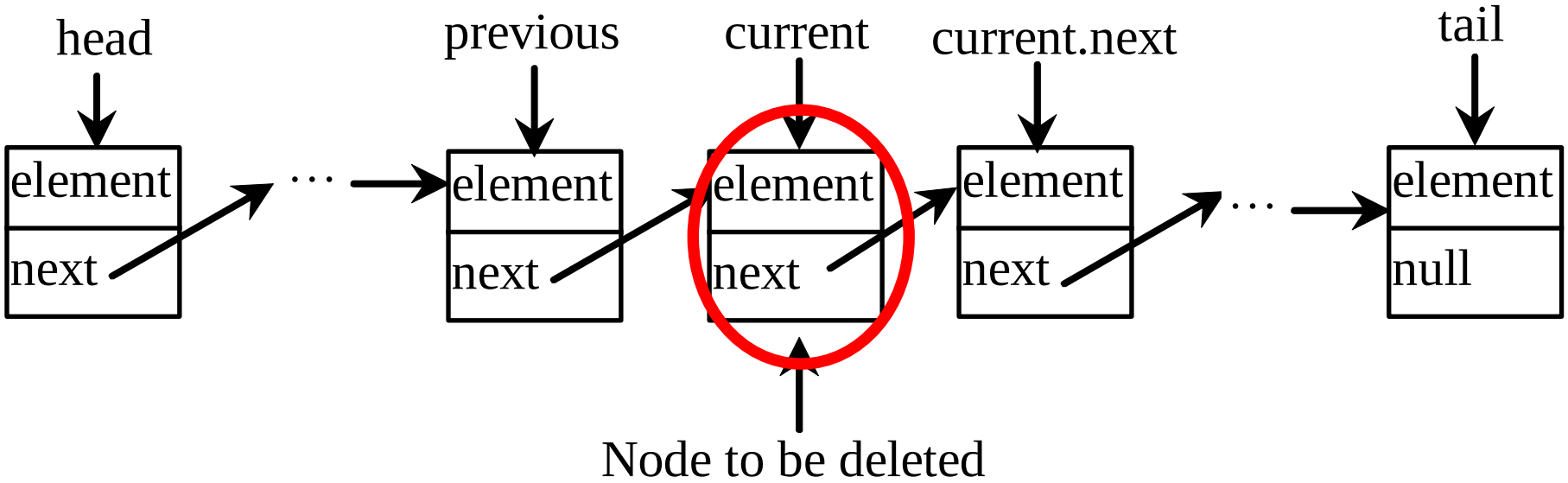
# Implementing `removeLast()`

```
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;
  }
}
```

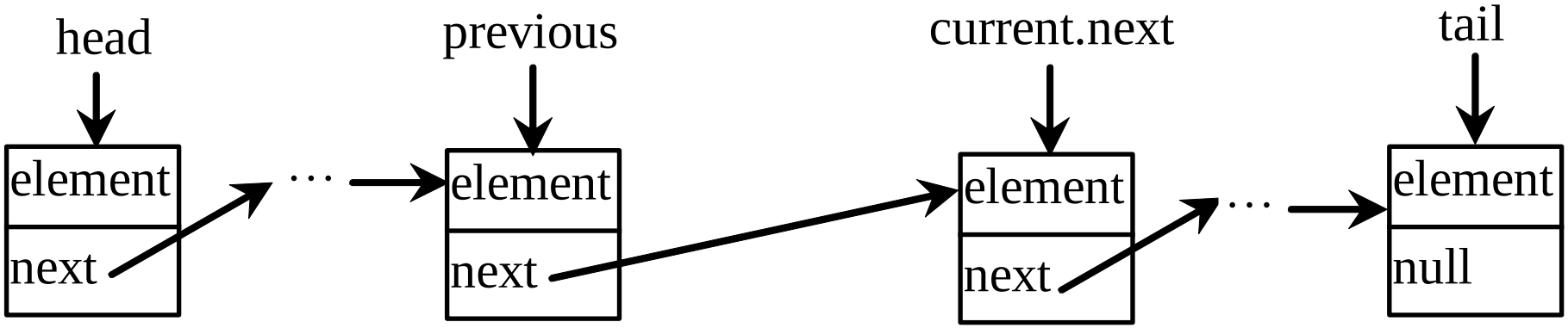


(c) Paul Fodor (CS Stony Brook) & Pearson

# Implementing `remove(int index)`




(a) Before the node is deleted.




(b) After the node is deleted.

# 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;
          // OR = (previous.next).next;
    size--;
    return current.element;
  }
}
```



(c) Paul Fodor (CS Stony Brook) & Pearson

# 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 or you can compare the current node with tail, or the current node with **`null`** to see that the list is empty)
    - 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;
}
```



(c) Paul Fodor (CS Stony Brook) & Pearson

```
// Complete MyLinkedList implementation
public class MyLinkedList<E> extends MyAbstractList<E> {
  private Node<E> head, tail;

  private static class Node<E> {
    E element;
    Node<E> next;
    public Node(E element) {
      this.element = element;
    }
  }

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

  /** Create a list from an array of objects */
  public MyLinkedList(E[] objects) {
    super(objects);
  }
```



(c) Paul Fodor (CS Stony Brook) & Pearson

```
/** Return the head element in the list */
public E getFirst() {
  if (size == 0) {
    return null;
  } else {
    return head.element;
  }
}

/** Return the last element in the list */
public E getLast() {
  if (size == 0) {
    return null;
  } else {
    return tail.element;
  }
}
```



(c) Paul Fodor (CS Stony Brook) & Pearson

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

/** Add an element to the end of the list */
public void addLast(E e) {
  Node<E> newNode = new Node<E>(e); // Create a new for element e
  if (tail == null) {
    head = tail = newNode; // The 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++; // Increase size
}
```



(c) Paul Fodor (CS Stony Brook) & Pearson

```
/** Add a new element at the specified index in this list
 * The index of the head element is 0 */
public void add(int index, E e) {
  if (index == 0) {
    addFirst(e);
  } else if (index >= size) {
    addLast(e);
  } 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>(e);
    (current.next).next = temp;
    size++;
  }
}
```



(c) Paul Fodor (CS Stony Brook) & Pearson

```
/** Remove the head node and
 *  return the object that is contained in the removed
 * node. */
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;
  }
}
```



(c) Paul Fodor (CS Stony Brook) & Pearson

```
/** Remove the last node and
 * return the object that is contained in the removed node. */
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;
  }
}
```



(c) Paul Fodor (CS Stony Brook) & Pearson

```
/** Remove the element at the specified position in this list.
 *  Return the element that was removed from the list. */
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;
  }
}
```



(c) Paul Fodor (CS Stony Brook) & Pearson

```
/** Return true if this list contains the element o */
public boolean contains(E e) {
  Node<E> current = head;
  while(current != null) {
    if (current.element.equals(e))
      return true;
    current = current.next;
  }
  return false;
}

//public boolean contains(E e) {
//  Node<E> current = head;
//  for (int i = 0; i < size; i++) {
//    if (current.element.equals(e))
//      return true;
//    current = current.next;
//  }
//  return false;
//}
```



(c) Paul Fodor (CS Stony Brook) & Pearson

```
/** Return the element from this list at the index */
public E get(int index) {
  if(size <= index)
    return null;
  Node<E> current = head;
  for(int i=0; i<index; i++)
    current = current.next;
  return current.element;
}
/** Replace the element at the specified position in this list
 *  with the specified element. */
public E set(int index, E e) {
  if(index >= size)
    return null;
  Node<E> current = head;
  for(int i=0; i<index; i++)
    current = current.next;
  E old = current.element;
  current.element = e;
  return old;
 }
```



(c) Paul Fodor (CS Stony Brook) & Pearson

```
/** Return the index of the head matching element in this list.
 *  Return -1 if no match. */
public int indexOf(E e) {
  Node<E> current = head;
  int i = 0;
  while(current != null) {
    if (current.element.equals(e))
      return i;
    i++;
    current = current.next;
  }
  return -1;
}

/** Return the index of the last matching element in this list
 *  Return -1 if no match. */
public int lastIndexOf(E e) {
  Node<E> current = head;
  int i = 0, lastIndex = -1;
  while(current != null) {
    if (current.element.equals(e))
      lastIndex = i;
    i++;
    current = current.next;
  }
  return lastIndex;
}
```



(c) Paul Fodor (CS Stony Brook) & Pearson

```
@Override /** Override toString() to return elements in the list */
public String toString() {
  StringBuilder result = new StringBuilder("[");
  Node<E> current = head;
  for (int i = 0; i < size; i++) {
    result.append(current.element);
    current = current.next;
    if (current != null) {
     result.append(", "); // Separate two elements with a comma
    } else {
     result.append("]"); // Insert the closing ] in the string
    }
  }
  return result.toString();
}

/** Clear the list */
public void clear() {
  head = tail = null;
}
```



(c) Paul Fodor (CS Stony Brook) & Pearson

```
@Override
public Iterator<E> iterator() {
  return new LinkedListIterator();
}
private class LinkedListIterator implements java.util.Iterator<E>{
      Node<E> current = head;
      int index = 0;
      @Override
      public boolean hasNext() {
            return (current !=null);
      }
      @Override
      public E next() {
            Node<E> node = current;
            current = current.next;
            index++;
            return node.element;
      }
      @Override
      public void remove() {
            MyLinkedList.this.remove(index);
      }      // to get this object from the outer class
}
}
```



(c) Paul Fodor (CS Stony Brook) & Pearson

```
public class TestLinkedList {
  public static void main(String[] args) {
    // Create a list for strings
    MyLinkedList<String> list = new MyLinkedList<String>();

    // Add elements to the list
    list.add("America"); // Add it to the list
    System.out.println("(1) " + list);

    list.add(0, "Canada"); // Add it to the beginning of the list
    System.out.println("(2) " + list);

    list.add("Russia"); // Add it to the end of the list
    System.out.println("(3) " + list);

    list.addLast("France"); // Add it to the end of the list
    System.out.println("(4) " + list);

    list.add(2, "Germany"); // Add it to the list at index 2
    System.out.println("(5) " + list);

    list.add(5, "Norway"); // Add it to the list at index 5
    System.out.println("(6) " + list);
```



(c) Paul Fodor (CS Stony Brook) & Pearson

```
    list.add(0, "Poland"); // Same as list.addFirst("Poland")
    System.out.println("(7) " + list);

    // Remove elements from the list
    list.remove(0); // Same as list.remove("Australia") in this case
    System.out.println("(8) " + list);

    list.remove(2); // Remove the element at index 2
    System.out.println("(9) " + list);

    list.remove(list.size() - 1); // Remove the last element
    System.out.println("(10) " + list);
  }
}
```

```
    (1) [America]
    (2) [Canada, America]
    (3) [Canada, America, Russia]
    (4) [Canada, America, Russia, France]
    (5) [Canada, America, Germany, Russia, France]
    (6) [Canada, America, Germany, Russia, France, Norway]
    (7) [Poland, Canada, America, Germany, Russia, France, Norway]
    (8) [Canada, America, Germany, Russia, France, Norway]
    (9) [Canada, America, Russia, France, Norway]
    (10) [Canada, America, Russia, France]
```



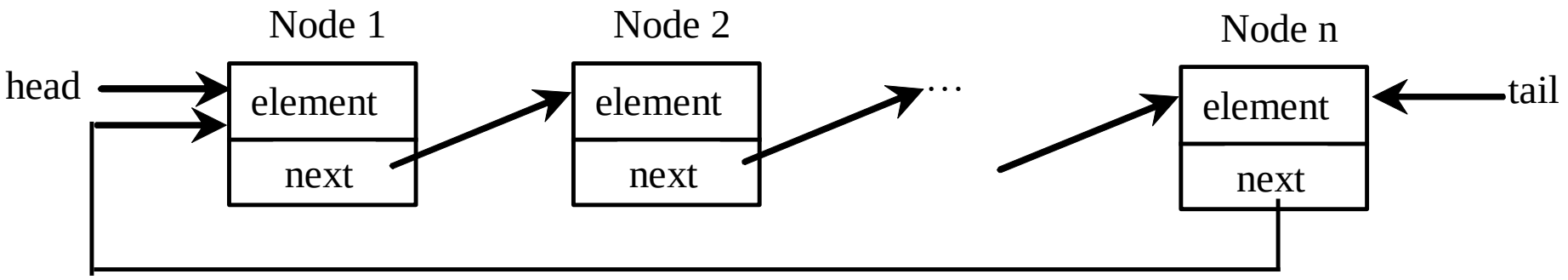
(c) Paul Fodor (CS Stony Brook) & Pearson

# Time Complexity for ArrayList vs LinkedList

| 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) | $O(1)$ | $O(n)$ |
| addFirst(e: E) | $O(n)$ | $O(1)$ |
| removeFirst() | $O(n)$ | $O(1)$ |

# Circular Linked Lists

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

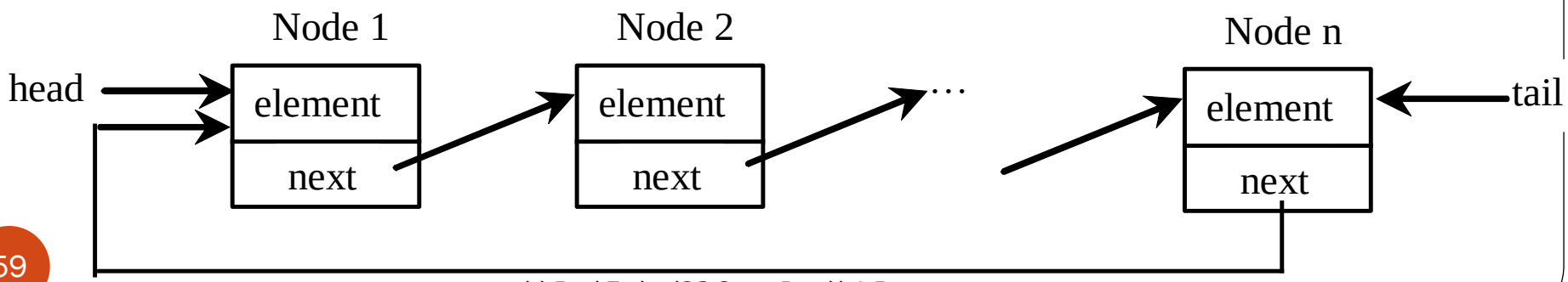





(c) Paul Fodor (CS Stony Brook) & Pearson

# Circular Linked Lists

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

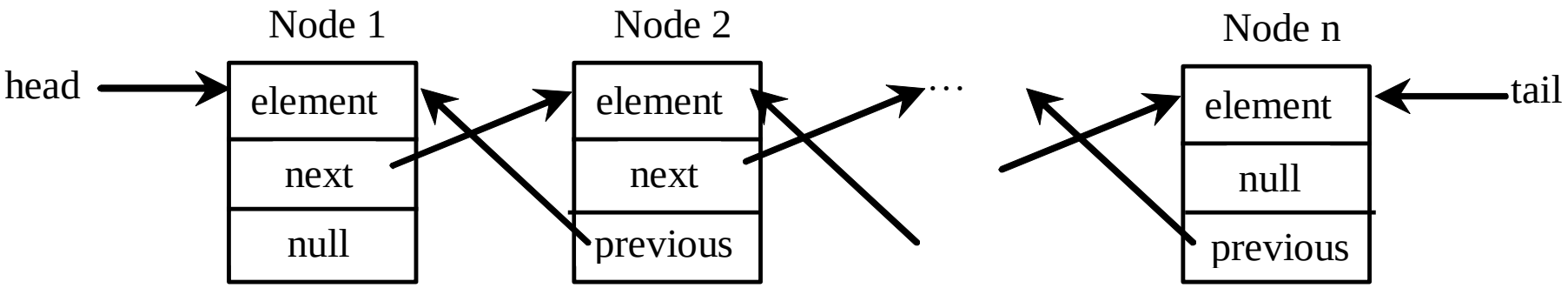





(c) Paul Fodor (CS Stony Brook) & Pearson

# 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***
  - A doubly linked list can be traversed ***forward*** and ***backward***

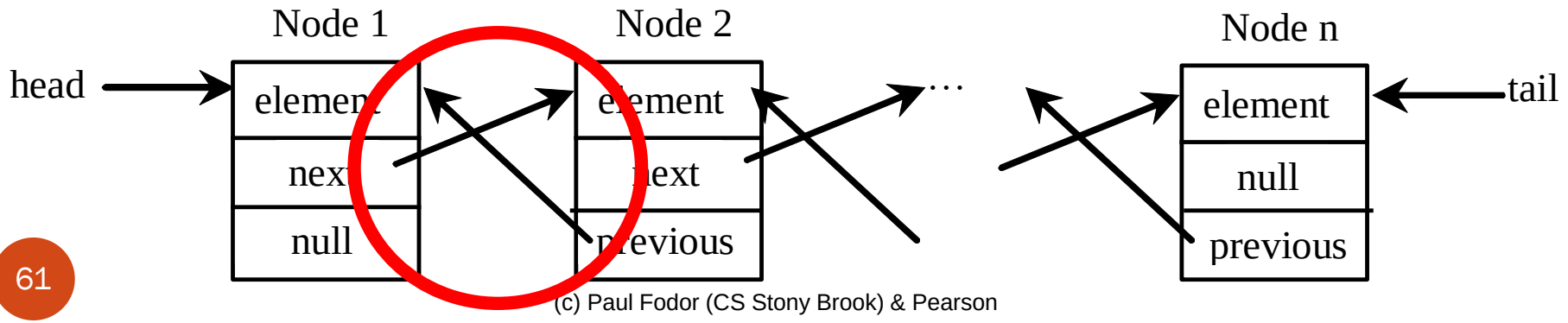





(c) Paul Fodor (CS Stony Brook) & Pearson

# Doubly Linked Lists

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

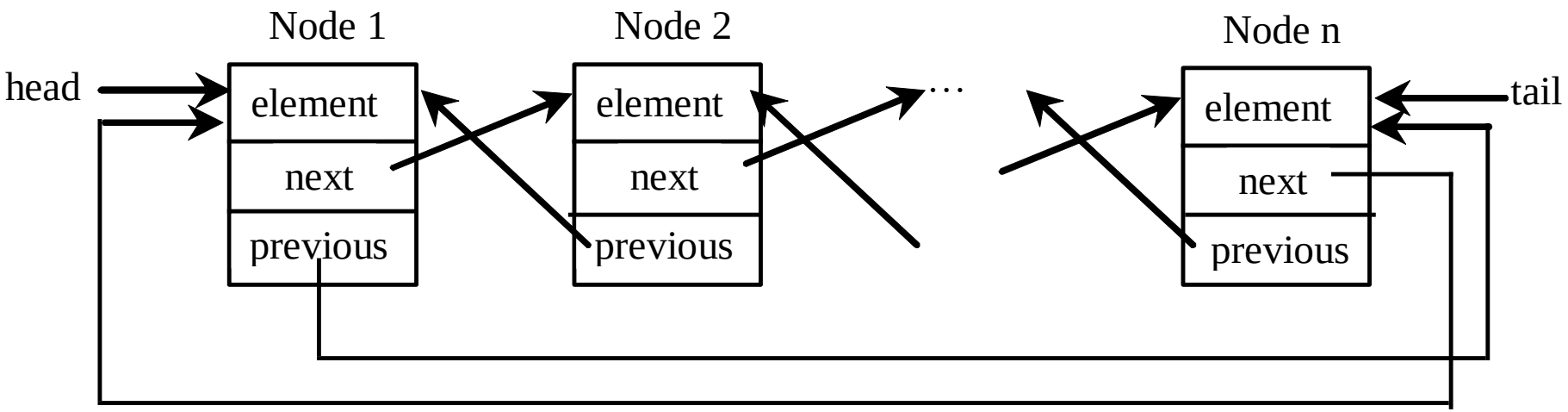





(c) Paul Fodor (CS Stony Brook) & Pearson

# Circular Doubly Linked Lists

- ***A circular doubly linked list*** is doubly linked list, with the exception 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:

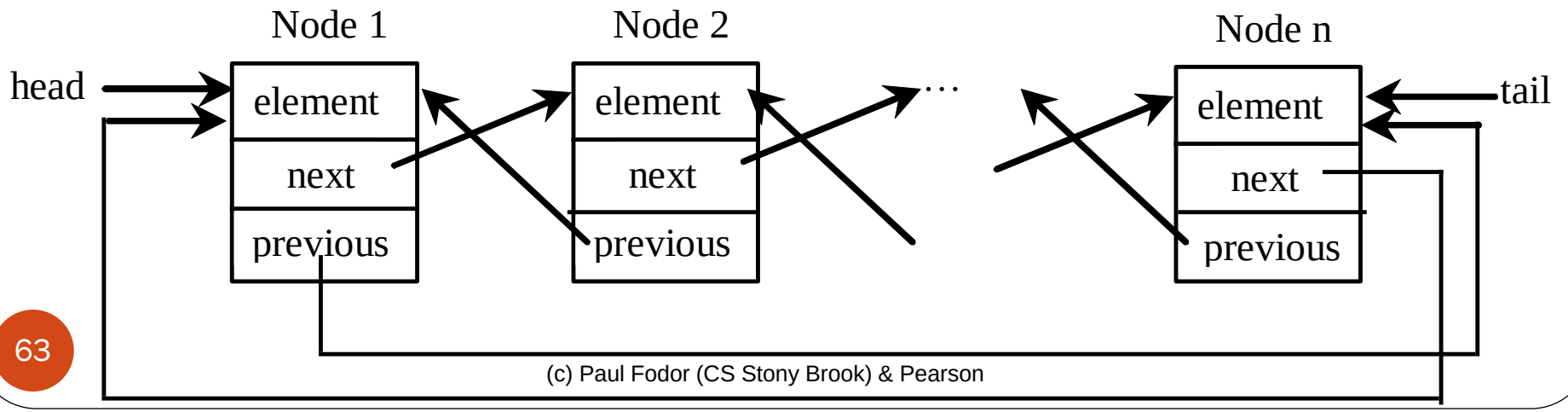





(c) Paul Fodor (CS Stony Brook) & Pearson

# Circular Doubly Linked Lists

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






(c) Paul Fodor (CS Stony Brook) & Pearson

# Same Time Complexity for CircularLinkedLists, DoubleLinkedLists and CircularDoubleLinkedLists

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

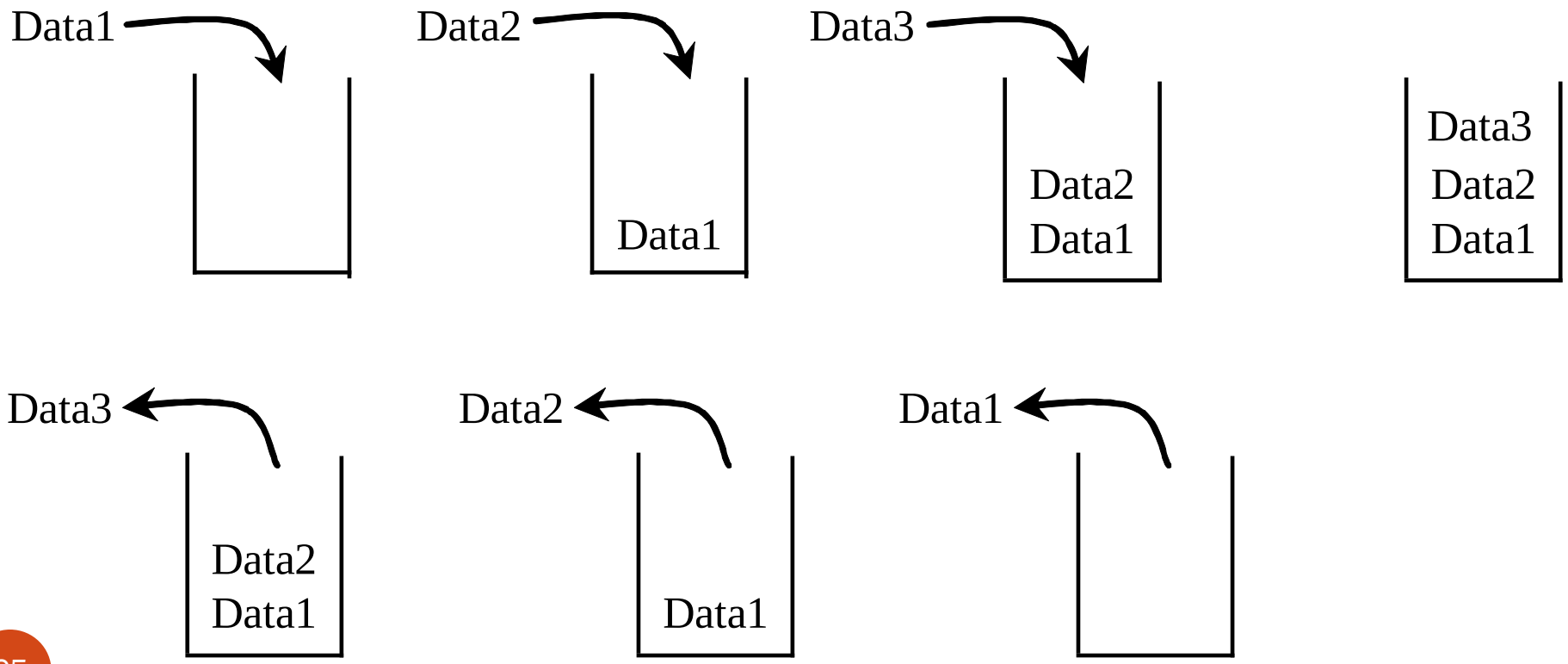
Better complexity only for: removeLast O(1) for DoubleLinkedLists as opposed to O(n) for LinkedLists



(c) Paul Fodor (CS Stony Brook) & Pearson

# 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

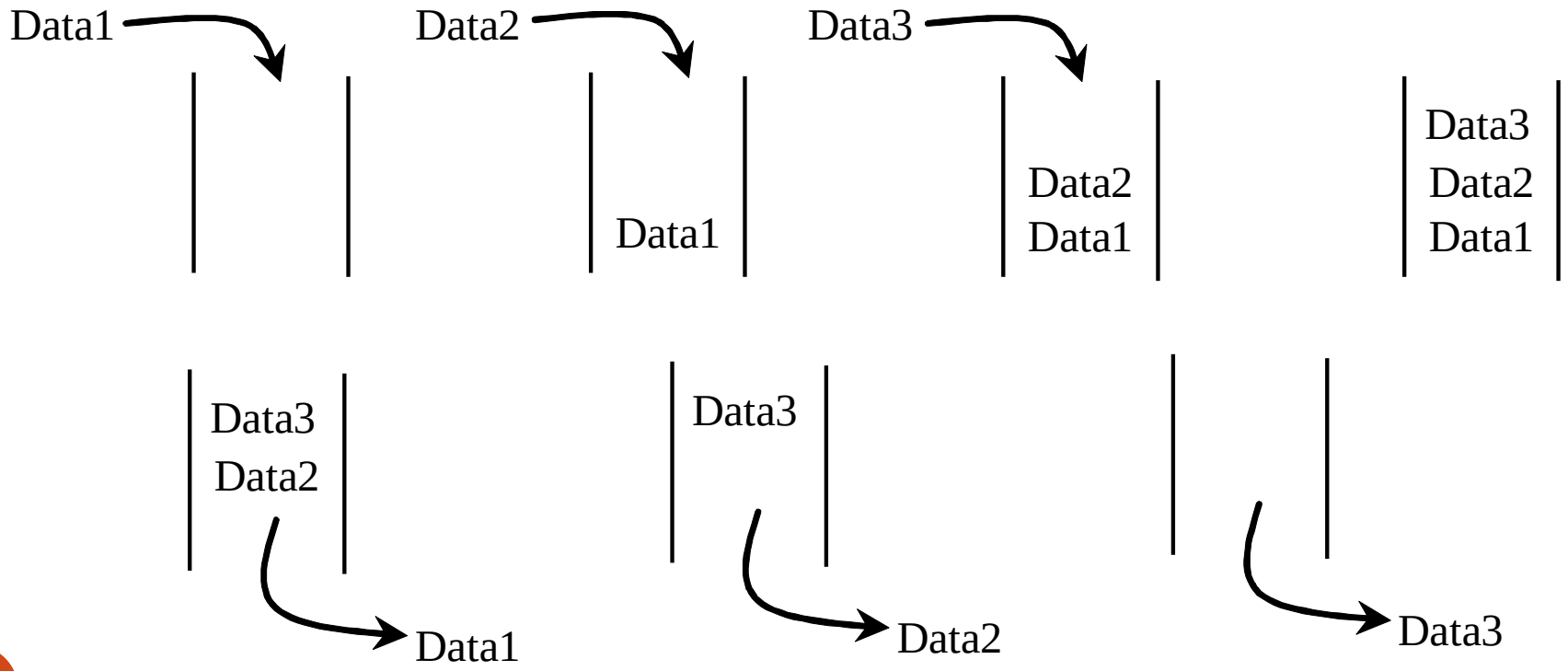





(c) Paul Fodor (CS Stony Brook) & Pearson

# Queues

- 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
  - Example: the customer orders at Starbucks






(c) Paul Fodor (CS Stony Brook) & Pearson

# Implementing Stacks and Queues

- Use an ArrayList to implement Stack
  - Since the insertion and deletion operations on a stack are made only at the **end** of the list, using an array list to implement a stack is more efficient than a linked list (removeLast is more efficient for ArrayLists)
- Use a LinkedList to implement Queue
  - Since **deletions are made at the beginning** of the list and **insertion is made at the end**, it is more efficient to implement a queue using a linked list than an array list



(c) Paul Fodor (CS Stony Brook) & Pearson

# Design of the Stack and Queue Classes

- There are two ways to design the stack and queue classes:
  - Using **inheritance**: define the stack class by extending the array list class, and the queue class by extending the linked list class


(a) Using inheritance

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


(b) Using composition



(c) Paul Fodor (CS Stony Brook) & Pearson

# Composition is Better

- Both designs are fine, but using composition is better because it enables you to define a completely new Stack and Queue classes **without inheriting the unnecessary and inappropriate methods from the array list and linked list**



(c) Paul Fodor (CS Stony Brook) & Pearson

# GenericStack and GenericQueue

| 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: LinkedList<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. |



(c) Paul Fodor (CS Stony Brook) & Pearson

```
public class GenericStack<E> {
  private java.util.ArrayList<E> list= new java.util.ArrayList<>();

  public E peek() {
    return list.get(getSize() - 1);
  }

  public void push(E o) {
    list.add(o);
  }

  public E pop() {
    E o = list.get(getSize() - 1);
    list.remove(getSize() - 1);
    return o;
  }
  public int getSize() {
    return list.size();
  }
  public boolean isEmpty() {
    return list.isEmpty();
  }
  @Override
  public String toString() {
    return "stack: " + list.toString();
  }
}
```



(c) Paul Fodor (CS Stony Brook) & Pearson

```
public class GenericQueue<E> {
  private java.util.LinkedList<E> list =
      new java.util.LinkedList<E>();

  public void enqueue(E e) {
    list.addLast(e);
  }

  public E dequeue() {
    return list.removeFirst();
  }

  public int getSize() {
    return list.size();
  }

  @Override
  public String toString() {
    return "Queue: " + list.toString();
  }
}
```



(c) Paul Fodor (CS Stony Brook) & Pearson

# Example: Using Stacks and Queues

- A program that creates a stack and a queue:

```
public class TestStackQueue {
  public static void main(String[] args) {
    // Create a stack
    GenericStack<String> stack = new GenericStack<>();

    // Add elements to the stack
    stack.push("Tom"); // Push it to the stack
    System.out.println("(1) " + stack);

    stack.push("Susan"); // Push it to the the stack
    System.out.println("(2) " + stack);

    stack.push("Kim"); // Push it to the stack
    stack.push("Michael"); // Push it to the stack
    System.out.println("(3) " + stack);

    // Remove elements from the stack
    System.out.println("(4) " + stack.pop());
    System.out.println("(5) " + stack.pop());
    System.out.println("(6) " + stack);
```

```
(1) stack: [Tom]
(2) stack: [Tom, Susan]
(3) stack: [Tom, Susan, Kim,
    Michael]
(4) Michael
(5) Kim
(6) stack: [Tom, Susan]
```



(c) Paul Fodor (CS Stony Brook) & Pearson

```
    // Create a queue
    GenericQueue<String> queue = new GenericQueue<>();

    // Add elements to the queue
    queue.enqueue("Tom"); // Add it to the queue
    System.out.println("(7) " + queue);

    queue.enqueue("Susan"); // Add it to the queue
    System.out.println("(8) " + queue);

    queue.enqueue("Kim"); // Add it to the queue
    queue.enqueue("Michael"); // Add it to the queue
    System.out.println("(9) " + queue);

    // Remove elements from the queue
    System.out.println("(10) " + queue.dequeue());
    System.out.println("(11) " + queue.dequeue());
    System.out.println("(12) " + queue);
  }
}
```

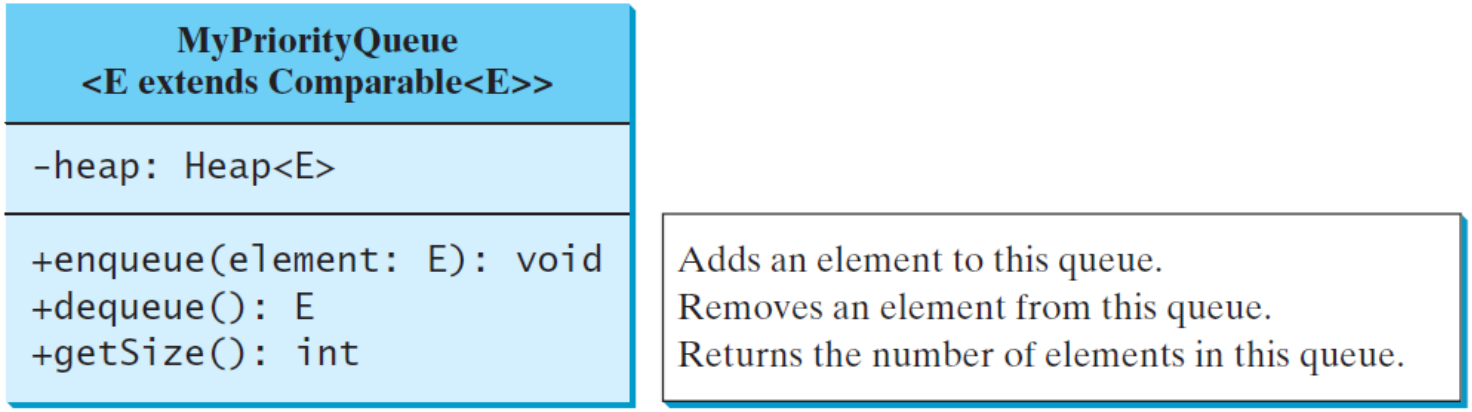
```
(7) Queue: [Tom]
(8) Queue: [Tom, Susan]
(9) Queue: [Tom, Susan, Kim, Michael]
(10) Tom
(11) Susan
(12) Queue: [Kim, Michael]
```



(c) Paul Fodor (CS Stony Brook) & Pearson

# Priority Queue

- A regular queue is a **first-in and first-out data** (***FIFO***) structure:
  - Elements are appended to the end of the queue and are removed from the beginning of the queue
- In a priority queue, elements are assigned with priorities
  - When accessing elements, the element with the highest priority is removed first, i.e., largest-in, first-out behavior
  - For example, the emergency room in a hospital assigns patients with priority numbers; the patient with the highest priority is treated first






(c) Paul Fodor (CS Stony Brook) & Pearson

```
public class MyPriorityQueue<E extends Comparable<E>> {
  private Heap<E> heap = new Heap<E>();

  public void enqueue(E newObject) {
    heap.add(newObject);
  }

  public E dequeue() {
    return heap.remove();
  }

  public int getSize() {
    return heap.getSize();
  }
}
```



(c) Paul Fodor (CS Stony Brook) & Pearson

```
public class TestPriorityQueue {
  public static void main(String[] args) {
    Patient patient1 = new Patient("John", 2);
    Patient patient2 = new Patient("Jim", 1);
    Patient patient3 = new Patient("Tim", 5);
    Patient patient4 = new Patient("Cindy", 7);

    MyPriorityQueue<Patient> priorityQueue =
       new MyPriorityQueue<>();

    priorityQueue.enqueue(patient1);
    priorityQueue.enqueue(patient2);
    priorityQueue.enqueue(patient3);
    priorityQueue.enqueue(patient4);

    while (priorityQueue.getSize() > 0)
      System.out.print(priorityQueue.dequeue() + " ");
  }

// Cindy(priority:7) Tim(priority:5) John(priority:2) Jim(priority:1)
```



(c) Paul Fodor (CS Stony Brook) & Pearson

```
  static class Patient implements Comparable<Patient> {
    private String name;
    private int priority;
    // patients with the same priority: first-in, first-out
    private int order;
    static int counter = 100;

    public Patient(String name, int priority) {
      this.name = name;
      this.priority = priority;
      this.order = counter--;
    }

    @Override
    public String toString() {
      return name + "(priority:" + priority + ")";
    }

    public int compareTo(Patient o) {
      if(this.priority > o.priority)
        return 1;
      else if(this.priority == o.priority)
        return this.order - o.order;
      else
        return -1;
    }
  }
}
```



(c) Paul Fodor (CS Stony Brook) & Pearson