OKANAGAN

COSC 121

Computer Programming II

Implementing Lists, Stacks, and Queues

Part 2/2

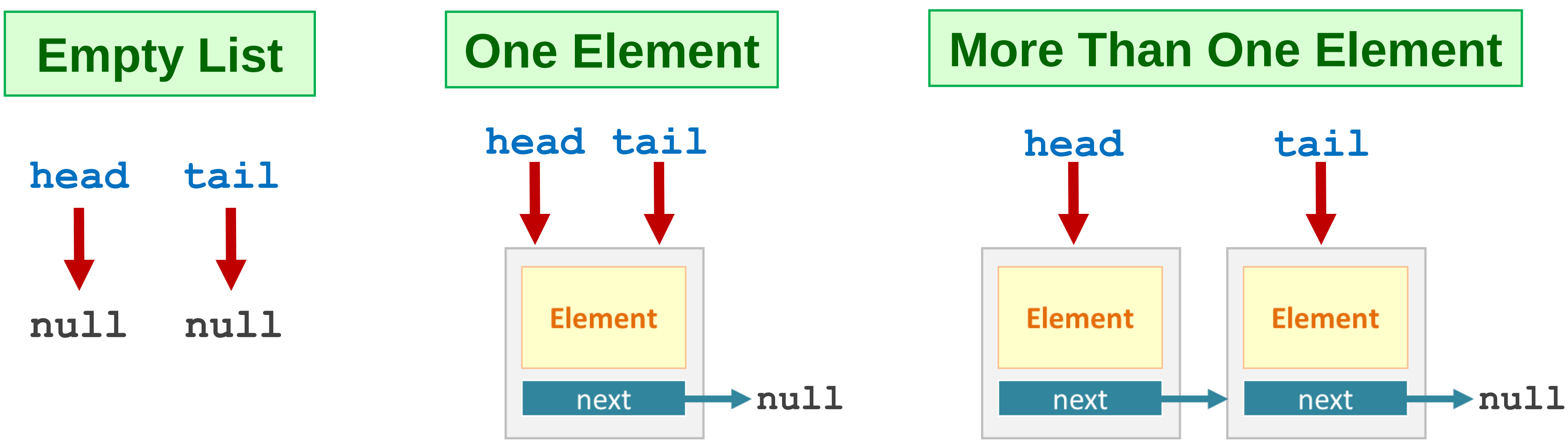
Dr. Mostafa Mohamed

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

Implementing Linked Lists *cont'd*

Remember: how to build a LinkedList !

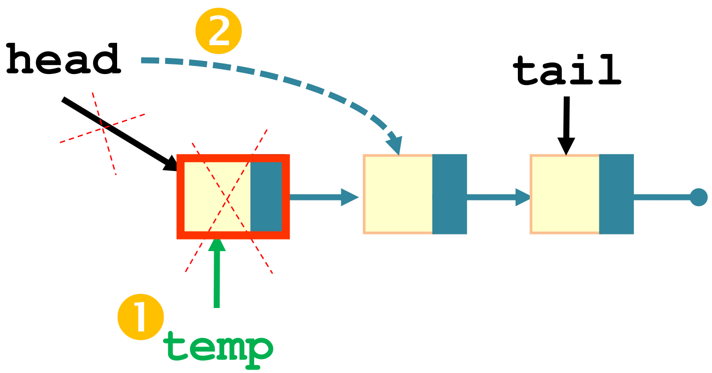
- `next`, `head`, `tail` are of the type *Node*
- A node is deleted if no references point to it
- you have to consider all three cases shown below:



REMOVING

removeFirst()

- **If empty list:**
  - **return null**
- **else**
  - Link a **temp** ref to first element (to return it)
  - Have **head** point to second node
  - If list becomes empty (i.e. head == null)
    - **Have tail point to null**
  - *decrement size*
  - return **temp.*element***

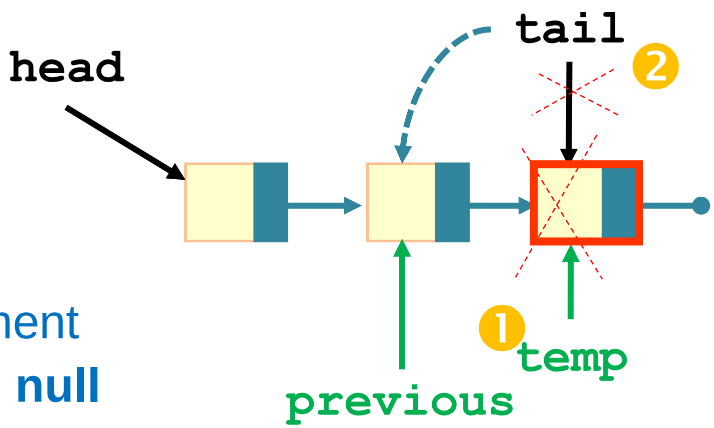



```
public E removeFirst() {
  if(head == null)
     return null;          // or throw NoSuchElementException
  Node<E> temp = head; // temp.element will be returned
  head = head.next;
  if(head == null)      // if list had only one node
     tail = null;          // now it is empty
  size--;
  return temp.element;
}
```

REMOVING

removeLast()

- **If empty** list → return null
- **else if 1** element → removeFirst
- **else**
  - Link a ***temp*** ref to last element (to return it)
  - Get a reference ***previous*** to 2nd to last element
  - Link **tail** to **previous**, and **previous.next** to **null**
  - return ***temp.element***

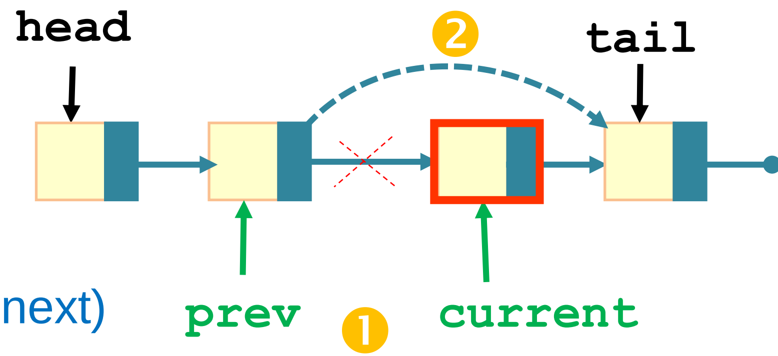



```
public E removeLast() {
   if(tail == null) return null;   // or throw NoSuchElementException
   else if (size == 1)             // if one element
       return removeFirst();
   else {
       Node<E> temp = tail;        // #1 in above diagram (will be returned)
       Node<E> previous = head;    // ]
       for(int i=0; i<size-2; i++) // ] get reference to 2nd to last node
           previous=previous.next; // ]
       tail = previous;            // #2 in above diagram
       tail.next = null;
       size--;
       return temp.element;
   }
}
```

REMOVING

remove(int index)

- **If index < 0 or ≥ size → return null** (this includes empty case where size=0)
- **Else if index = 0 → removeFirst**
- **Else if index = size-1 → removeLast**
- **Else**
  - Get 2 references ***prev & current*** to the element at *index-1* and *index*
  - Remove the element (prev.next = current.next)
  - return **current.element**




```
public E remove(int index) {
    if(index<0 || index>=size) return null; // or throw exception
    else if(index == 0)        return removeFirst();
    else if(index == size-1)   return removeLast();
    else {
        Node<E> previous = head;               // ]
        for(int i=0; i<index-1; i++)           // ]-get reference to previous node
            previous = previous.next;          // ]
        Node<E> current = previous.next;       // node to be removed
        previous.next = current.next;          // unlink node
        size--;
        return current.element;
    }
}
```

GETTING

Implement `getFirst()` , `getLast()` , `get(int index)`

`getFirst()`

- **If empty list → return null**
- **Else →** return **head.element**

`getLast()`

- **If empty list → return null**
- **Else →** return **tail.element**

```
public E getFirst() {
    if (size == 0)
        return null;
    else
        return head.element;
}
public E getLast() {
    if (size == 0)
        return null;
    else
        return tail.element;
}
```

GETTING

get(index i)

- **If index < 0 or ≥ size → return null** (this includes empty case where size=0)
- **Else if index = 0 → getFirst**
- **Else if index = size-1 → getLast**
- **Else**
  - Get a reference ***current*** to the element at ***index***
  - ***Return current.element***

```
public E get(int index){
    if(index < 0 || index >= size) return null;
    else if(index == 0) return getFirst();
    else if(index == size-1) return getLast();
    else {
        Node<E> current = head;          // ]
        for (int i = 0; i < index; i++) // ]- get a reference to index
            current = current.next;     // ]
        return current.element;
    }
}
```

The ITERATOR

Here we need to implement the iterator of our MyLinkedList.

- 1) Declare a method `iterator` in `MyLinkedList`

```
public java.util.Iterator<E> iterator() {
    return new MyListIterator();
}
```

- 2) create a class MyListIterator the has the required methods

```
private class MyListIterator implements java.util.Iterator<E> {
    private Node<E> current = head; // Current index
    public boolean hasNext() {
        return (current != null);
    }
    public E next() {
        E e = current.element;
        current = current.next;
        return e;
    }
    public void remove() {
        System.out.println("Implementation left as an exercise");
    }
}
```

MyLinkedList

```java
public class MyLinkedList<E> {
  // ATTRIBUTES
  private int size=0;
  private Node<E> head = null, tail = null;
  //METHODS:
  // isEmpty
  public boolean isEmpty(){return (size==0);}
  // size
  public int size(){return size;}

  //ADDING
  // addFirst (add node then increment size)
  public void addFirst(E element){
     Node<E> n = new Node<E>(element);
     if(isEmpty())
        head = tail = n;
     else{
        n.next = head;
        head = n;
     }
     size++;
  }
  // addLast (add node then increment size)
  public void addLast(E element){
     Node<E> n = new Node<E>(element);
     if(isEmpty())
        head = tail = n;
     else{
        tail.next = n;
        tail = n;   //tail = tail.next;
     }
     size++;
  }
  // add(index,e)
  public void add(int index, E element){
     if(index < 0 || index > size)
        throw new IndexOutOfBoundsException();
     else if(index == 0)
        addFirst(element);
     else if(index == size)
        addLast(element);
     else{
        Node<E> node = new Node<E>(element);
        //get a reference to element at index-1
        Node<E> current = head;
        for (int i = 0; i < index-1; i++)
           current = current.next;
        node.next = current.next;
        current.next = node;
     }
     size++;
  }
  // add(e)
  public void add(E element){
     addLast(element);
  }
```

```java
//REMOVING
// removeFirst and decrement size
public E removeFirst(){
   if(isEmpty())
      return null;
      //OR throw new NoSuchElementException();
   else{
      Node<E> temp = head;
      head = head.next;
      if(head == null) tail = null;
      size--;
      return temp.element;
   }
}
// remove last and decrement size
public E removeLast(){
   if(isEmpty())
      return null;
      //OR throw new NoSuchElementException();
   else if(size == 1)
      return removeFirst();
   else{ //more than one element
      //temp save last node in order to return it
      Node<E> temp = tail;
      //get a reference to node at size-2
      Node<E> current = head;
      for (int i = 0; i < size-2; i++)
         current = current.next;
      //move tail to current
      tail = current;
      tail.next = null;
      size--;
      //return last node
      return temp.element;
   }
}
// remove by index
public E remove(int index){
   if(index < 0 || index > size-1)
      throw new IndexOutOfBoundsException();
   else if(index == 0)
      return removeFirst();
   else if (index == size-1)
      return removeLast();
   else{
      Node<E> prev = head;
      for (int i = 0; i < index-1; i++)
         prev = prev.next;
      Node<E> current = prev.next;
      prev.next = current.next;
      size--;
      return current.element;
   }
}
```

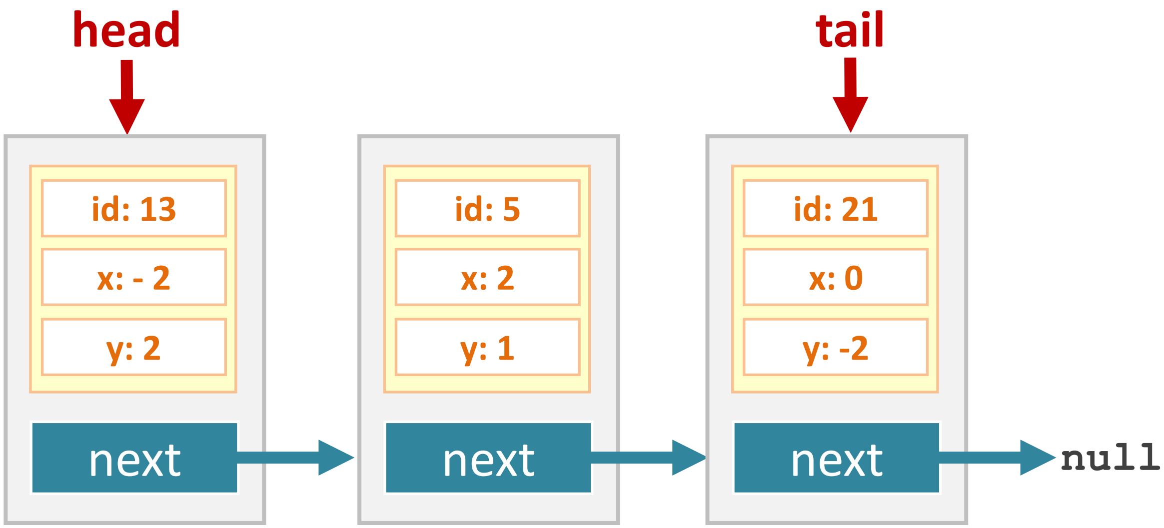
```java
  //GET/SET
  // getFirst/getLast
  public E getFirst(){
     if(isEmpty())
        return null;
     else
        return head.element;
  }
  public E getLast(){
     if(isEmpty())
        return null;
     else
        return tail.element;
  }

 //ITERATOR
 //iterator
  public Iterator<E> iterator(){
     return new MyIterator();
  }
  // Iterator class
  class MyIterator implements Iterator<E>{
     private Node<E> current = head;
     public boolean hasNext() {
        return (current != null);
     }
     public E next() {
        E tmp = current.element;
        current = current.next;
        return tmp;
     }
  }

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

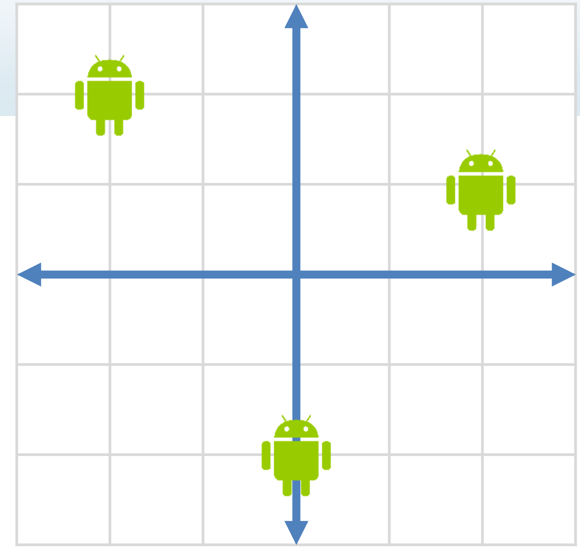
Testing MyLinkedList

```
public class MyLinkedListTest {
   public static void main(String[] args) {
       MyLinkedList<Integer> a = new MyLinkedList<Integer>();
       //fill the list
       for (int i = 0; i < 12; i++) {
           a.add(i);
       }
       displayList(a);

       a.add(3, 17);
       a.addFirst(19);
       a.addLast(21);
       displayList(a);

       //remove some elements by index
       a.remove(1);   // remove element at index 1
       a.remove(2);   // remove element at index 2
       a.removeFirst();
       a.removeLast();
       displayList(a);
   }

   public static void displayList(MyLinkedList<Integer> a){
       Iterator<Integer> itr = a.iterator();
       while(itr.hasNext())
           System.out.print(itr.next()+" ");
       System.out.println();
   }
}
```

Practice

Given the linked list, `RobotsList`, which maintains information about the position of a group of robots.





```java
public class Node {
    Robot robot;
    Node next;
    public Node(Robot r){
        robot = r;
    }
}
```

```java
public class RobotsList {
    private Node head = null, tail = null;
    private int size = 0;
    public void addFirst(Robot robot) {...}
    public void addLast(Robot e) {...}
    public void add(int index, Robot robot) {...}
    public Robot removeFirst() {...}
    public Robot removeLast() {...}
    public Robot remove(int index) {...}
}
```

```java
public class Robot {
    private Integer id, x, y;
    public Robot(Integer id,Integer x,Integer y){...}
    public Integer getId() {...}
    public Integer getX() {...}
    public Integer getY() {...}
    public void setId(Integer id) {...}
    public void setX(Integer x) {...}
    public void setY(Integer y) {...}
    public String toString(){...}
}
```

Practice, cont.

Write the following methods:

```
Integer getX(Integer id) //for the first occurrence of id
Integer getY(Integer id)
void setLocation(Integer id, Integer x, Integer y)
```

- get/set the location based on a given id

```
void printAllIn(Integer x1,Integer y1,Integer x2,Integer y2)
```

- print the id's of all robots in a given area defined by (x1,y1) to (x2,y2). For simplicity, **assume x1<x2 and y1<y2**

```
Integer count(Integer x1,Integer y1,Integer x2,Integer y2)
```

- get the number of robots in a given area defined by (x1,y1) to (x2,y2). Assume x1<x2 and y1<y2.

```
void addAfter(Integer id, Robot robot)
```

- add a node after a node with a given id (first occurrence of id).

Practice, cont.

`boolean contains(Integer id)`

- Returns true if the id is in the list .

`boolean addUnique(Robot robot)`

- add a robot to the end of the list only when its id is not in the list (and return true), otherwise return false.

`Robot remove(Integer id)` *

- remove the first occurrence of a robot with a given id. If the id cannot be found, throw NoSuchElementException
  - **Challenging with singly linked list**
  - **Easy with doubly linked list**

`void removeAllIn(Integer x1,Integer y1,Integer x2,Integer y2)`

- remove all robots in a given area defined by (x1,y1) to (x2,y2). Assume x1<x2 and y1<y2.

Practice

Assuming that the robots instances stored in the `RobotsList` in the previous example are ordered ascendingly based on their ids. Write a methods to add another robot to the list such that the list **remains ordered**.

`void addOrdered(Robot robot)` → `challenging`

Implementing ArrayLists

ArrayLists

Array lists use fixed-size arrays.

- Whenever you need to **expand** the capacity, **create a new** larger array to replace the current array.

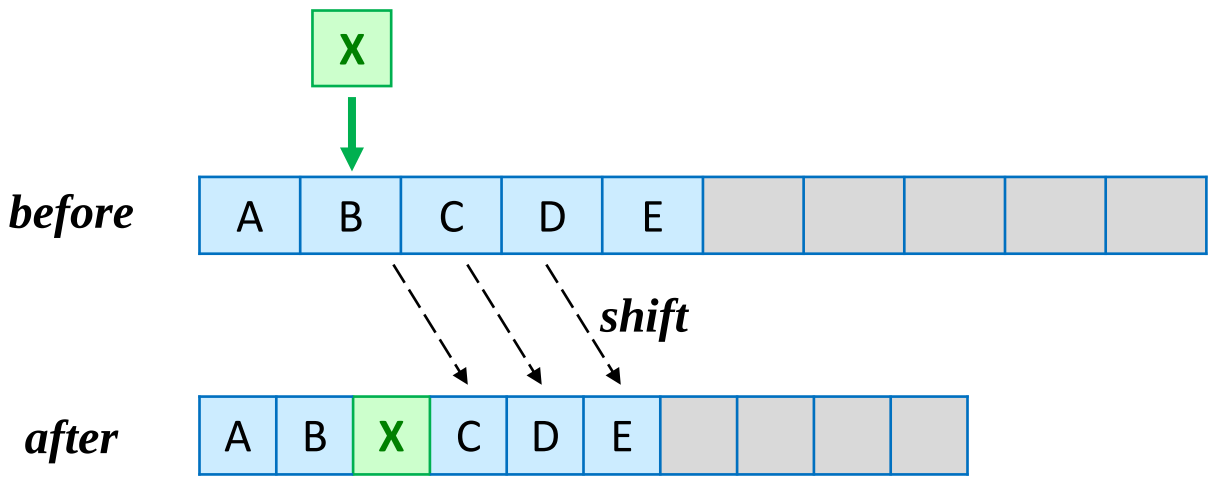
When inserting a new element into the array,

1. First, ensure there is **enough room** in the array.
2. If not, expand:
   a) Create a new array with the size as twice as the current one + 1.
   b) Copy the elements from the current array to the new array.
   c) The new array now becomes the current array.

Insertion

To insert a new element at a specified index:

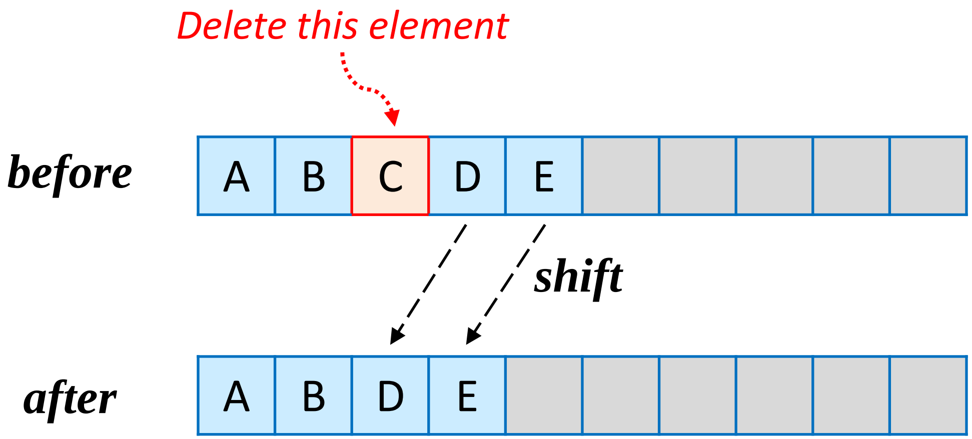
1) ensure there is enough room for new element (if not, expand)
2) shift all elements after the index to the right by one.
   - Obviously if you are inserting at the end, there will be no shifting.
3) insert the element.
4) increase size by 1.



Deletion

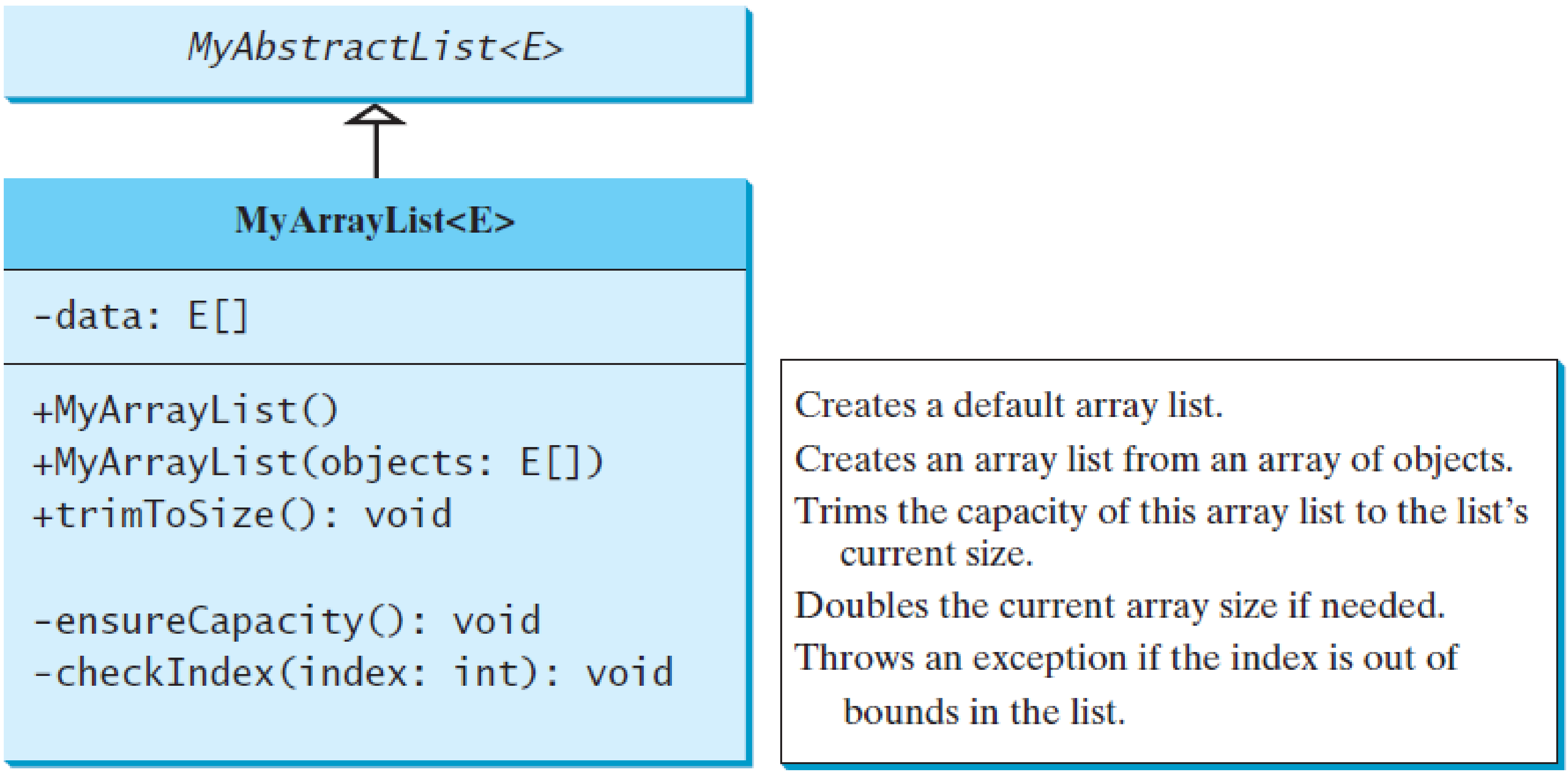
To remove an element at a specified index

1) shift all elements after the index to the left by one.
2) decrease size by 1.



Implementing MyArrayList

An array list uses an array of the generic type E, and has methods that allow inserting, deleting, etc. below shows some of these methods.



Ensuring there is Enough Room when Inserting

Lets say that our array list has a private variable `data`

```
private static final int INIT_CAPACITY = 10;
private E[] data = (E[]) new Object[INIT_CAPACITY];
```

Whenever you add one element, make sure there is enough space first, and expand as needed:

```
private void ensureCapacity() {
  if(size >= data.length) { //if array is full, expand!
    E[] newData = (E[]) new Object[data.length*2 + 1];
    System.arraycopy(data, 0, newData, 0, data.length);
    data = newData;
  }
}
```

Checking Validity of a Given Index

If you want to check the validity of a given index:

- For all methods except adding methods:
  - index must be from **`0`** to **`size-1`** inclusive.

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

- For add methods
  - Index must be from **`0`** to **`size`** inclusive
    - Note that index could be equal to size which means add after last element

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

ADDING

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

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

Implement `add(int index, E e)`

- 1) ensure there is enough space. Also check the index.
- 2) move the elements 1 step to the right after the specified element
- 3) insert the new element

```
public void add(int index, E e) {
    checkIndexForAdd(index);
    ensureCapacity();
    // Move the elements to the right after the specified index
    for (int i = size - 1; i >= index; i--)
        data[i + 1] = data[i];
    // Insert new element to data[index]
    data[index] = e;
    // Increase size by 1
    size++;
}
```

REMOVING

Implement **`remove(int index)`**

Algorithm:

- Get a reference to the element at index *(will be returned)*
- Shift any subsequent elements to the left
- Set last one to null.
- Return the element that was removed from the list.

```
public E remove(int index) {
    checkIndex(index);
    E temp = data[index];                       // to return it
    for (int i = index+1; i < size; i++)        // shift left
        data[i-1] = data[i];
    data[size-1]=null;                          // last element
    size--;
    return temp;
}
```

REMOVING

Implement `remove(E e)`

Algorithm:

- Search the list of the element.
- If found, remove it and return true
- Otherwise, return false

```
public boolean remove(E e) {
   for (int i = 0; i < size; i++)//search for the element
   if(data[i].equals(e)) {//if found,remove it and return true
      remove(i);
      return true;
   }
   return false;//if not found, return false
}
```

SETTING, GETTING

Implement: `void set(int index, E e), E get(int index)`

```
public E set(int index, E e) {
    checkIndex(index);
    E old = data[index];
    data[index] = e;
    return old;
}
public E get(int index) {
    checkIndex(index);
    return data[index];
}
```

contains, indexOf, and lastIndexOf

Implement

`boolean contains(E e),int indexOf(E e),int lastIndexOf(E e)`

```
public boolean contains(E e) {
    for (int i = 0; i < size; i++)
        if (e.equals(data[i]))
            return true;
    return false;
}
public int indexOf(E e) {
    for (int i = 0; i < size; i++)
        if (e.equals(data[i]))
            return i;

    return -1;
}
public int lastIndexOf(E e) {
    for (int i = size - 1; i >= 0; i--)
        if (e.equals(data[i]))
            return i;
    return -1;
}
```

clear and trimToSize

Implement: `void clear(), trimToSize()`

```
public void clear() {
    data = (E[]) new Object[INITIAL_CAPACITY];
    size = 0;
}
```

```
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
}
```

The ITERATOR

```
public java.util.Iterator<E> iterator() {
    return new MyIterator();
}
private class MyIterator implements java.util.Iterator<E> {
    private int current = 0; // Current index
    public boolean hasNext() {
        return (current < size);
    }
    public E next() {
        return data[current++];
    }
    public void remove() { //remove last element read by next
        MyArrayList.this.remove(current-1);
    }
}
```

MyArrayList

```java
public class MyArrayList<E> {
 //ATTRIBUTES
 private static final int INIT_CAP = 10;
 //int size, E[] data
 private int size = 0;
 private E[] data = (E[]) new
Object[INIT_CAP];

 //METHODS
 //size(), isEmpty()
 public int size(){return size;}
 public boolean isEmplty(){return size==0;}

 //add(index,e), add(e)
 public void add(int index, E e){
    checkIndexForAdd(index);
    ensureCapacity();
    for (int i = size; i > index; i--)
       data[i] = data[i-1];
    data[index] = e;
    size++;
 }
 public void add(E e){
    add(size, e); //add e to the end
 }
 //remove(index)
 public E remove(int index){
    checkIndex(index);
    E tmp = data[index];
    for (int i = index; i < size-1; i++)
       data[i] = data[i+1];
    data[size-1] = null;
    size--;
    return tmp;
 }

 // int remove(E e)
 // find the index i of first occurrence of e
 // remove(i, e) and return i.
 // if e cannot be found, then return -1
```

```java
//set(index,e)/get(index)
public E set(int index, E e){
   checkIndex(index);
   E tmp = data[index];
   data[index] = e;
   return tmp;
}
public E get(int index){
   checkIndex(index);
   return data[index];
}
//contains, indexOf, lastIndexOf
public boolean contains(E e){
   for (int i = 0; i < size; i++)
      if(data[i].equals(e))
         return true;
   return false;
}
public int indexOf(E e){
   for (int i = 0; i < size; i++)
      if(data[i].equals(e))
         return i;
   return -1;
}
public int lastIndexOf(E e){
   for (int i = size-1; i >=0; i--)
      if(data[i].equals(e))
         return i;
   return -1;
}
//clear
public void clear(){
   E[] newData = (E[]) new Object[INIT_CAP];
   data = newData;
   size = 0;
}
//trimToSize
public void trimToSize(){
   if(size < data.length){
      E[] newData = (E[]) new Object[size];
      System.arraycopy(data, 0, newData, 0, size);
      data = newData;
   }
}
```

```java
 //Iterator
 public Iterator<E> iterator(){
    return new MyIterator();
 }
 private class MyIterator implements
 Iterator<E>{
    private int current = 0;
    public boolean hasNext() {
       return (current<size);
    }
    public E next() {
       return data[current++];
    }
 }

 /* HELPER METHODS */
 private void ensureCapacity(){
    if(size >= data.length){  //if # of
elements == capacity then expand
       E[] newdata=(E[])new Object[2*size+1];
       System.arraycopy(data, 0, newdata,
            0, data.length);
       data = newdata;
    }
 }
 //checkIndex
 private void checkIndex(int index){
    if(index<0 || index >= size)
       throw new IndexOutOfBoundsException();
 }
 //checkIndexForAdd
 private void checkIndexForAdd(int index){
    if(index<0 || index > size)
       throw new IndexOutOfBoundsException();
 }

}
```

Testing MyArrayList

```java
public class MyArrayListTest {
   public static void main(String[] args) {
       MyArrayList<Integer> list = new MyArrayList<Integer>();

       // adding
       for (int i = 0; i <= 10; i++)
           list.add(i);
       displayList(list);
       list.add(3, 8);
       displayList(list);

       // removing
       list.remove(1); // remove element at index 1
       list.remove(2); // remove element at index 2
       displayList(list);
   }
   public static void displayList(MyArrayList<Integer> list) {
       Iterator<Integer> itr = list.iterator();
       while (itr.hasNext())
           System.out.print(itr.next() + " ");
       System.out.println();
   }
}
```

Practice

1) If you change the code in the `ensureCapacity` from

- E[] **new**Data = (E[])(**new** Object[size * **2** + **1**]);

to

- E[] **new**Data = (E[])(**new** Object[size * **2**]);

the program is incorrect. Can you find the reason?

Hint: consider an ArrayList after calling clear() then trimToSize()

2) Add a method to MyArrayList with the following header:

int removeDuplicates()

The method should remove all duplicates from the array list. Assume that you have a constructor that MyArrayList(int n) where n is the initial number of elements.

Stacks & Queues

Design of the Stack and Queue Classes

One way to implement them is as follows:

- Stack → use an array list
  - *We practiced on that before!*
  - www.cs.armstrong.edu/liang/animation/web/Stack.html

- Queue → use a linked list
  - Since **deletions are made at the beginning** of the list, it is more efficient to implement a queue using a linked list
  - www.cs.armstrong.edu/liang/animation/web/Queue.html

Design of the Stack and Queue Classes

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

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

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

Composition is better!

- because it enables you to define a complete new stack or queue class **without inheriting unnecessary and inappropriate** methods from the array/linked list.

Remember: Stacks

You have seen before how to implement a stack using array lists

GenericStack<E>	
-list: ArrayList<E>	A list to store elements.
+GenericStack()	**Generates 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 this stack is empty.**

```
import java.util.ArrayList;
public class MyStack<E> {
  private ArrayList<E> list = new ArrayList<E>();
  public boolean isEmpty() { return list.isEmpty(); }
  public int getSize()     { return list.size(); }
  public E peek()          { return list.get(getSize()-1); }
  public E pop()           { return list.remove(getSize()-1); }
  public void push(E e)    { list.add(e); }
}
```

Queues

Similarly, you can implement a queue using a linked lists

GenericQueue<E>
-list: LinkedList<E>
+enqueue(e: E): void
+dequeue(): E
+getSize(): int

Adds an element to this queue.

Removes an element from this queue.

Returns the number of elements from this queue.

Why LinkedList is used, not ArrayList?

```
public class MyQueue<E> {
    private MyLinkedList<E> list = new MyLinkedList<E>();

    public void enqueue(E e)  { list.addLast(e); }
    public E dequeue()        { return list.removeFirst(); }
    public int getSize()      { return list.size(); }
}
```