IT 179
7

Introduction to Lists

Introduction to Lists

- A **list** is a collection of elements, each with a position or index
- *Iterators* facilitate sequential access to lists
- Classes `ArrayList`, `Vector`, and `LinkedList` are *subclasses* of abstract class `AbstractList` and *implement* the `List` interface

List Interface and ArrayList Class

- An array is an indexed structure
- In an indexed structure,
  - elements may be accessed in any order using subscript values
  - elements can be accessed in sequence using a loop that increments the subscript
- With a Java array, you **cannot**
  - increase or decrease its length (length is fixed)
  - add an element at a specified position without shifting elements to make room
  - remove an element at a specified position and keep the elements contiguous without shifting elements to fill in the gap

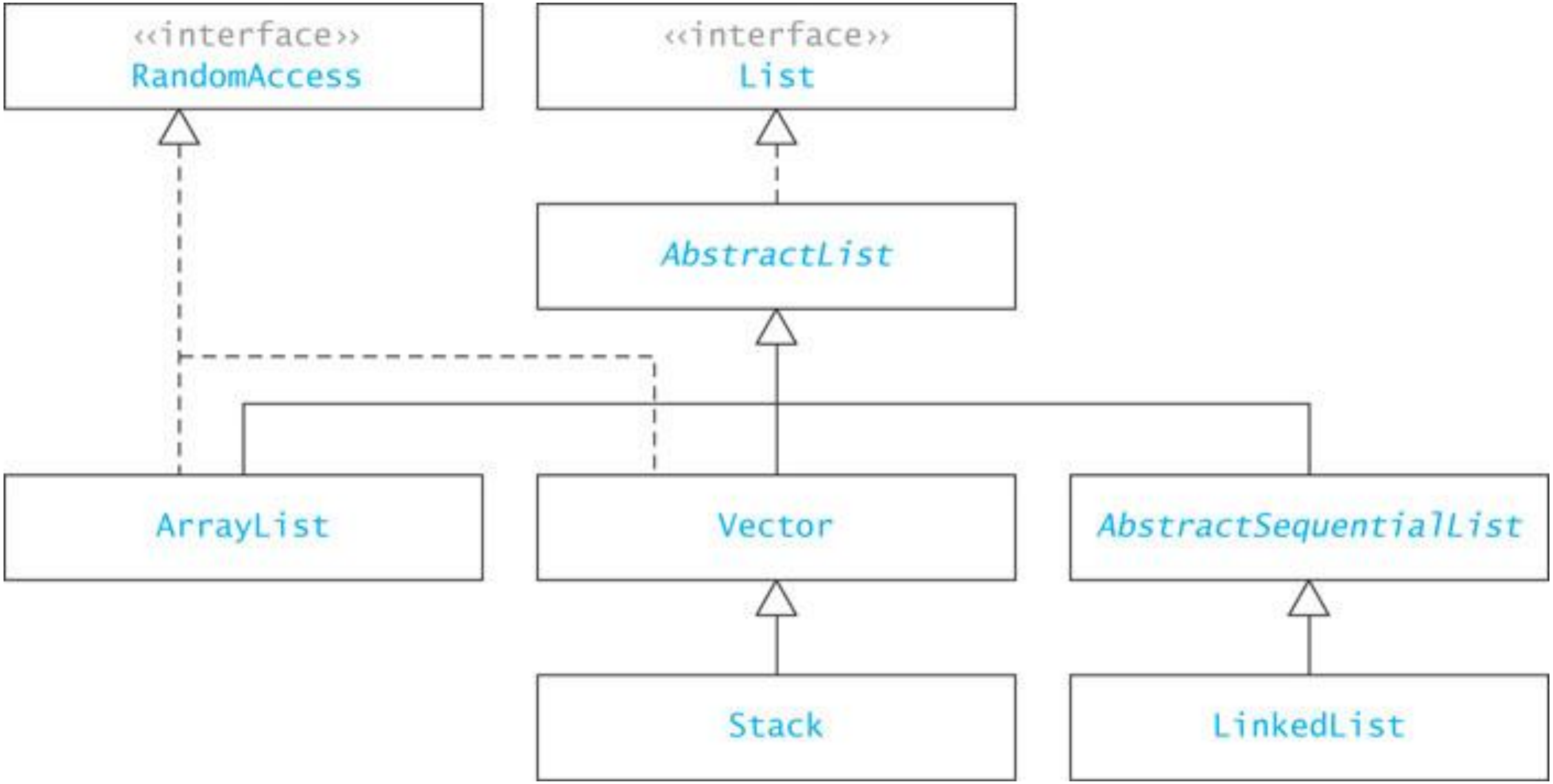
List Interface and ArrayList Class (cont.)

- Java provides a **List** interface as part of its API java.util
- Classes that implement the List interface provide the functionality of an indexed data structure and offer many more operations
- A sample of the operations:
  - Retrieve an element at a specified position (method get)
  - Replace an element at a specified position (method set)
  - Find a specified target value (method indexOf)
  - Add an element at either end (method add)
  - Remove an element (method remove)
  - Insert or remove an element at any position (method add)
  - Find the size of the list (method size)
  - Traverse the list structure without managing a subscript
- All classes introduced in this chapter support these operations, but they do not support them with the same degree of efficiency

Methods in the `List` Interface - E is a type parameter

Method	**Behavior**
`E get(int index)`	Returns the data in the element at position `index`
`E set(int index, E anEntry)`	Stores a reference to `anEntry` in the element at position `index`. Returns the data formerly at position `index`
`int size()`	Gets the current size of the `List`
`boolean add(E anEntry)`	Adds a reference to `anEntry` at the end of the List. Always returns `true`
`void add(int index, E anEntry)`	Adds a reference to `anEntry`, inserting it before the item at position `index`
`int indexOf(E target)`	Searches for `target` and returns the position of the first occurrence, or -1 if it is not in the `List`
`E remove (int index)`	Removes the entry formerly at position `index` and returns it

`java.util.List` Interface and Its Implementers



ArrayList al = new ArrayList();

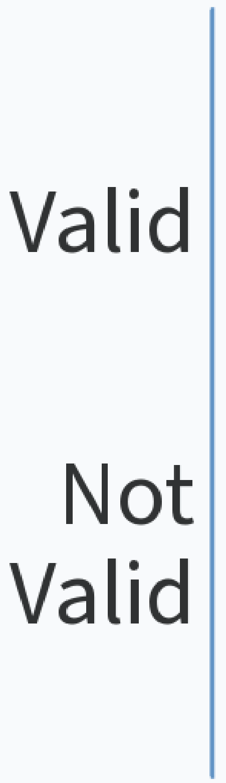
Valid

Not Valid

When poll is active, respond at **pollev.com/abdelmounaam190**

Text **ABDELMOUNAAM190** to **37607** once to join

ArrayList al = new List();



Powered by Poll Everywhere

When poll is active, respond at **pollev.com/abdelmounaam190**

Text **ABDELMOUNAAM190** to **37607** once to join

List al = new ArrayList();

List al = new List();

Valid

Not Valid

List Interface and ArrayList Class

- □ Unlike the Array data structure, classes that implement the List interface cannot store primitive types
- □ Classes must store values as **objects**

```
List<float> myList = new ArrayList<float>();
```
❌

- □ This requires you to wrap primitive types, such as int and double in object wrappers, in this case, Integer and Double

`List` Interface and `ArrayList` Class

✗ `List<float> myList = new ArrayList<float>();`

✓ `List<Float> myList = new ArrayList<Float>();`

✗ `List<double> myList = new ArrayList<double>();`

✓ `List<Double> myList = new ArrayList<Double>();`

`ArrayList` Class

- The simplest class that implements the List interface
- An improvement over an array object
- Use when:
  - you will be adding new elements to the end of a list
  - you need to access elements quickly in any order

ArrayList Class (cont.)

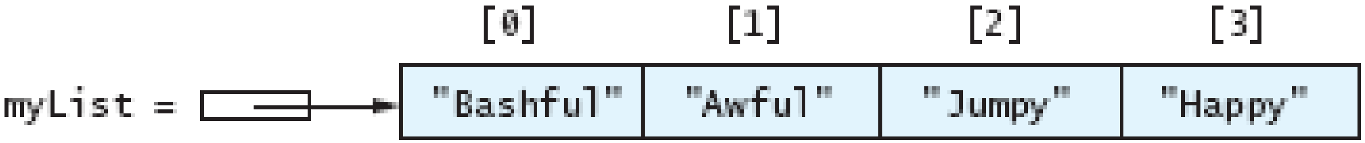
- To declare a List object whose elements will reference String objects:

```
List<String> myList = new ArrayList<String>();
```

- The initial ArrayList is empty and has a default initial capacity of 10 elements
- To add strings to the list,

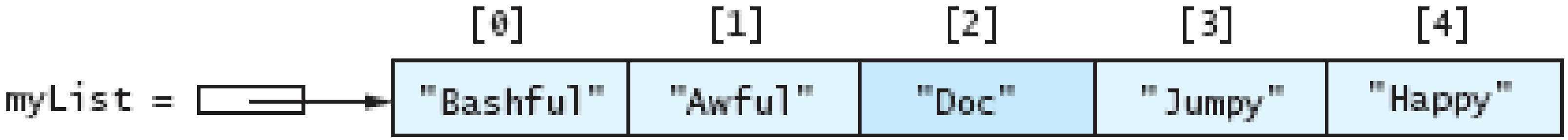
```
myList.add("Bashful");
myList.add("Awful");
myList.add("Jumpy");
myList.add("Happy");
```

`ArrayList` **Class** (cont.)



- □ Adding an element with subscript 2:

```
myList.add(2, "Doc");
```



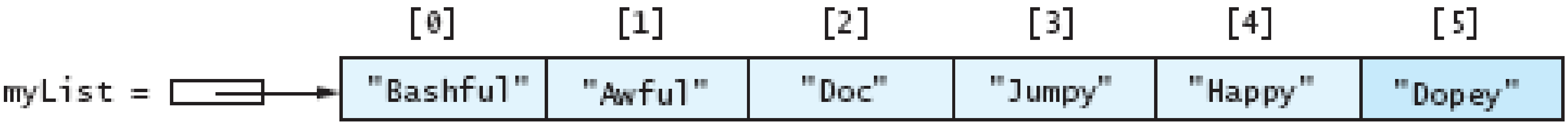
After insertion of "Doc" before the third element

- □ Notice that the subscripts of `"Jumpy"` and `"Happy"` have changed from [2],[3] to [3],[4]

ArrayList Class (cont.)

- When no subscript is specified, an element is added at the end of the list:

```
myList.add("Dopey");
```

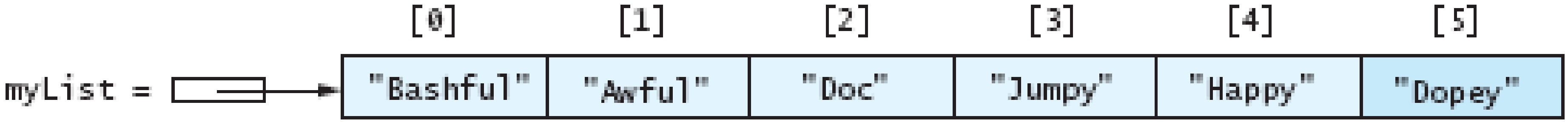


[0]	[1]	[2]	[3]	[4]	[5]
"Bashful"	"Awful"	"Doc"	"Jumpy"	"Happy"	"Dopey"

After insertion of "Dopey" at the end

`ArrayList` **Class** (cont.)

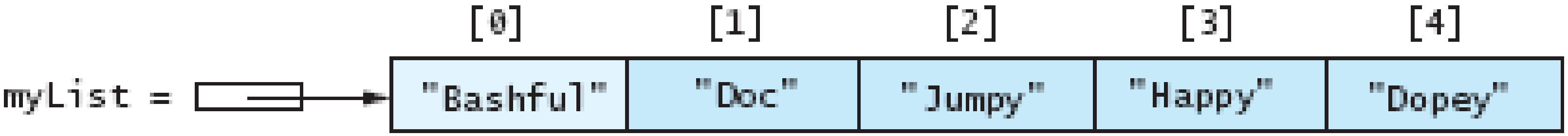
- Removing an element:



myList =

[0]	[1]	[2]	[3]	[4]	[5]
"Bashful"	"Awful"	"Doc"	"Jumpy"	"Happy"	"Dopey"

```
myList.remove(1);
```



myList =

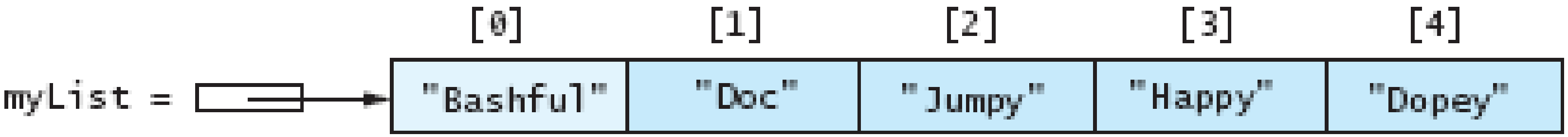
[0]	[1]	[2]	[3]	[4]
"Bashful"	"Doc"	"Jumpy"	"Happy"	"Dopey"

After removal of "Awful"

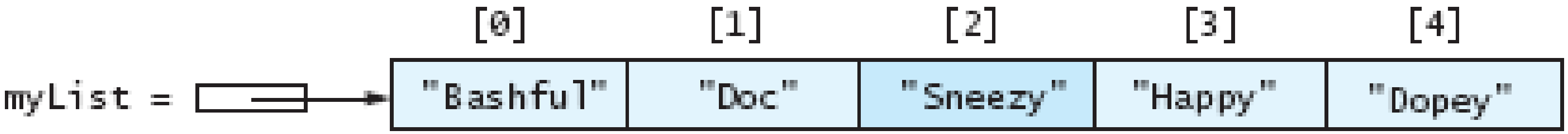
- The subscripts strings referenced by [2] to [5] have changed to [1] to [4]

`ArrayList` Class (cont.)

- You may also replace an element:

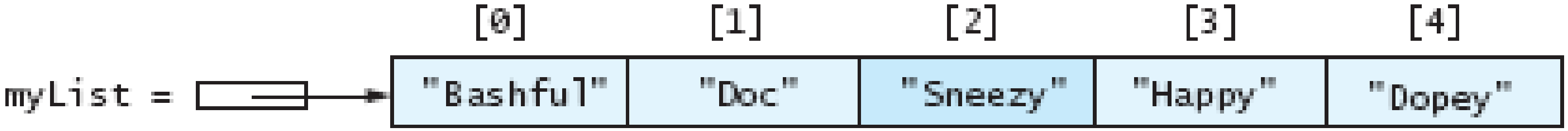


```
myList.set(2, "Sneezy");
```



After replacing "Jumpy" with "Sneezy"

ArrayList Class (cont.)

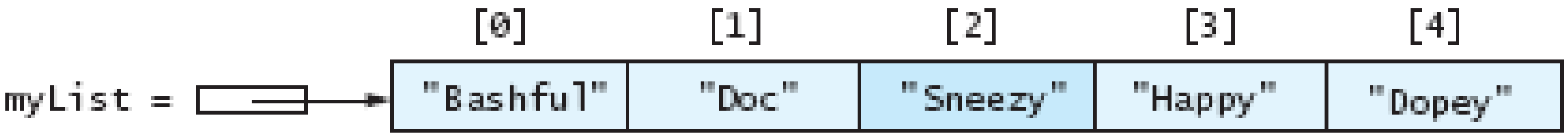


- You cannot access an element using a bracket index as you can with arrays (array[1])
- Instead, you must use the get() method:

```
String dwarf = myList.get(2);
```

- The value of dwarf becomes "Sneezy"

ArrayList Class (cont.)



- You can also search an ArrayList:

```
myList.indexOf("Sneezy");
```

- This returns 2 while

```
myList.indexOf("Jumpy");
```

- returns -1 which indicates an unsuccessful search

Generic Collections

- The statements

```
List<String> myList = new ArrayList<String>();
   List<Integer> myInts = new ArrayList<>();
    var myFamily = new ArrayList<People>();
```

use a language feature called *generic collections* or *generics*

- The second statement uses the *diamond operator* `<>` to reduce redundancy
- The third statement uses the keyword `var` (introduced in Java 10) to simplify declarations when data type can be implied
- All 3 statements creates a `List` of objects of a specified type (`String`, `Integer`, or `People`); only references of the specified type - can be stored in the list
- The type parameter sets the data type of all objects stored in a collection

Generic Collections (cont.)

- The general declaration for generic collection is

```
CollectionClassName<E> variable = new CollectionClassName<>();
```

- The <E> indicates a type parameter
- Adding a noncompatible type to a generic collection will generate an error during compile time
- However, primitive types will be autoboxed:

```
ArrayList<Integer> myList = new ArrayList<>();
myList.add(Integer.valueOf(3)); // ok
myList.add(3); // also ok! 3 is automatically wrapped
                  in an Integer object
myList.add(new String("Hello")); // generates a type
                                    incompatability error
```

Applications of `ArrayList`

Section 2.3

Example Application of `ArrayList`

- Use of for each to access array elements in sequence

```
var someInts = new ArrayList<>();
int[] nums = {5, 7, 2, 15};
// Load ArrayList someInts from nums
for (int numNext : nums) {
    someInts.add(numNext);
}
```

`numNext` **is an** `int`**; it is automatically wrapped in an** `Integer` **object**

- Use of for each to access objects in an ArrayList in sequence

```
int sum = 0;
for (Integer sumNext : someInts) {
    sum += sumNext;
}
```

`sumNext` **is unboxed and its** `int` **value is added to** `sum`

Phone Directory Application

```
public class DirectoryEntry {
   String name;
   String number;
}
```

Create a class for objects stored in the directory

Phone Directory Application (cont.)

```
public class DirectoryEntry {
  String name;
  String number;
}

private ArrayList<DirectoryEntry> theDirectory =
          new ArrayList<>();
```

Create the directory

Adding an object to the Directory

```
public class DirectoryEntry {
  String name;
  String number;
}


private ArrayList<DirectoryEntry> theDirectory =
          new ArrayList<>();


theDirectory.add(new DirectoryEntry("Jane Smith",
                                    "555-1212"));
```

Append a new `DirectoryEntry` **object to the directory**

Retrieving an entry if in the directory

```
int index = theDirectory.indexOf(new DirectoryEntry(aName,
                                                     ""));
if (index != -1)
  dE = theDirectory.get(index);
else
  dE = null;
```

Method indexOf searches theDirectory by applying the equals method for class DirectoryEntry. Assume DirectoryEntry's equals method compares name fields

dE references the directory entry with name aName.

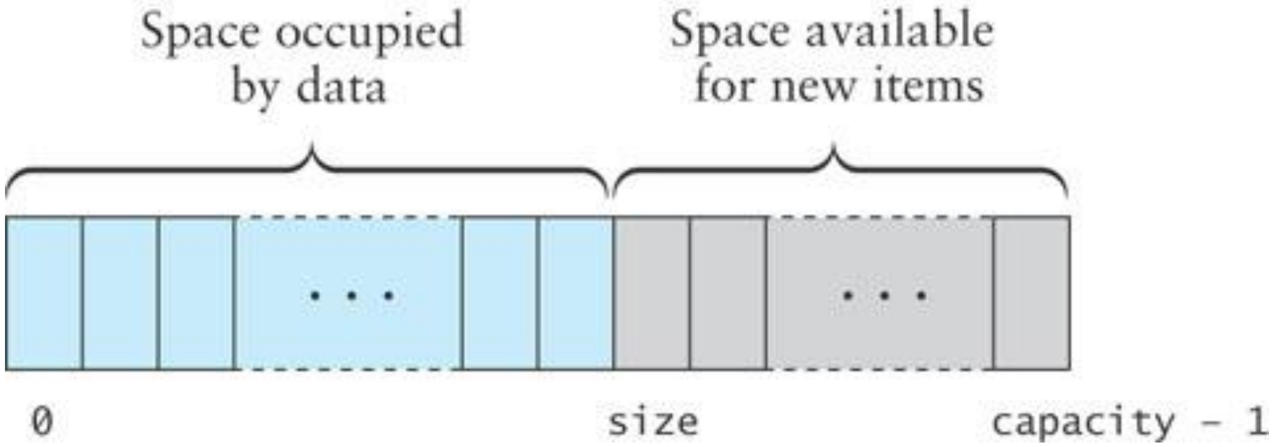
If aName is not found, dE is null.

Implementation of an `ArrayList` Class

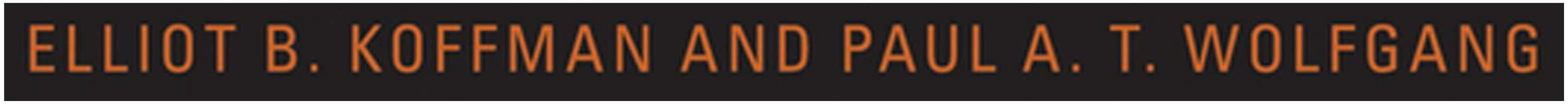
Section 2.4

Implementing an `ArrayList` Class

- `KWArrayList`: a simple implementation of `ArrayList`
  - Physical size of array indicated by data field capacity
  - Number of data items indicated by the data field size



Why KWArrayList?



ELLIOT B. KOFFMAN AND PAUL A. T. WOLFGANG

KWArrayList Fields

```
import java.util.*;


/** This class implements some of the methods of the Java
ArrayList class
*/
public class KWArrayList<E> {
  // Data fields
  /** The default initial capacity */
  private static final int INITIAL_CAPACITY = 10;
  /** The underlying data array */
  private E[] theData;
  /** The current size */
  private int size = 0;
  /** The current capacity */
  private int capacity = 0;
}
```

`KWArrayList` Constructor

```
public KWArrayList () {
    capacity = INITIAL_CAPACITY;
    theData = (E[]) new Object[capacity];
}
```

This statement allocates storage for an array of type `Object` and then casts the array object to type E[]

Although this may cause a compiler warning, it's fine

Implementing `ArrayList.add(E)`

- □ We will implement two add methods

- □ One will append at the end of the list

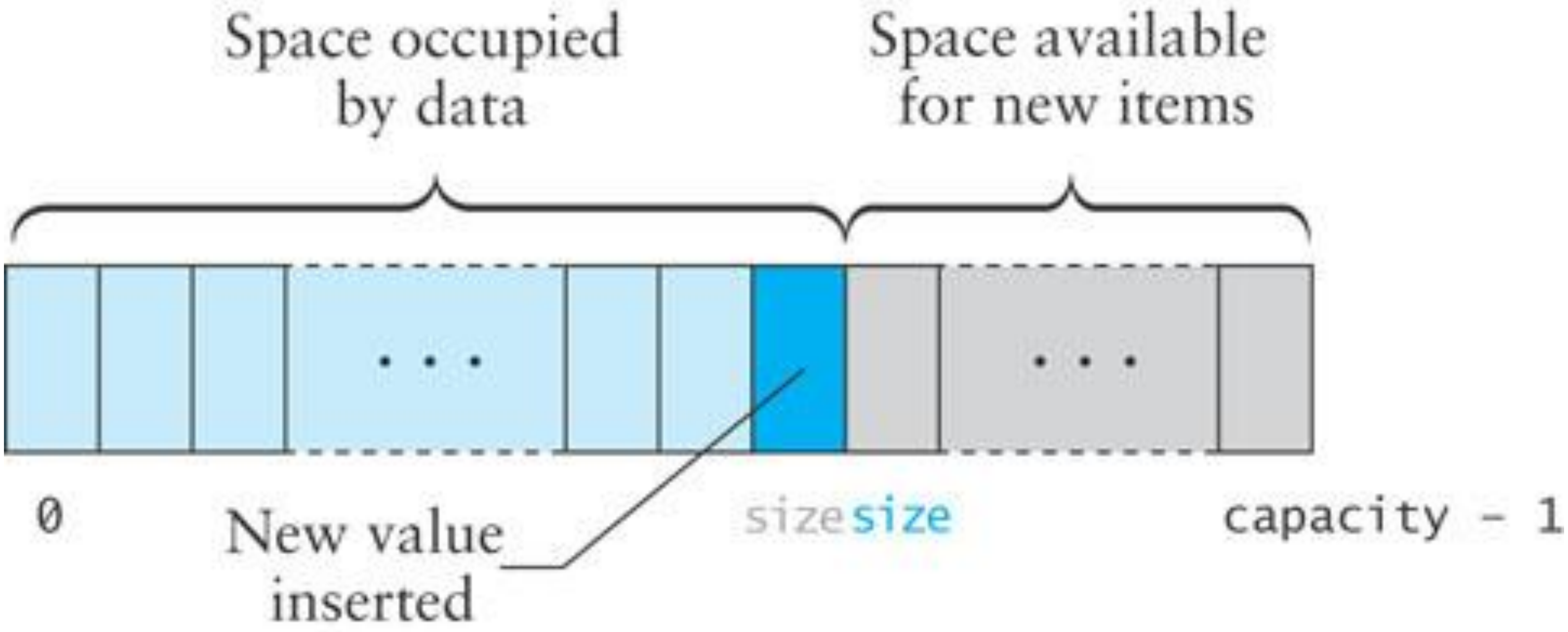
  `add(E e)`

- □ The other will insert an item at a specified position

  add(int index, E element)

Implementing `ArrayList.add(E)` (cont.)

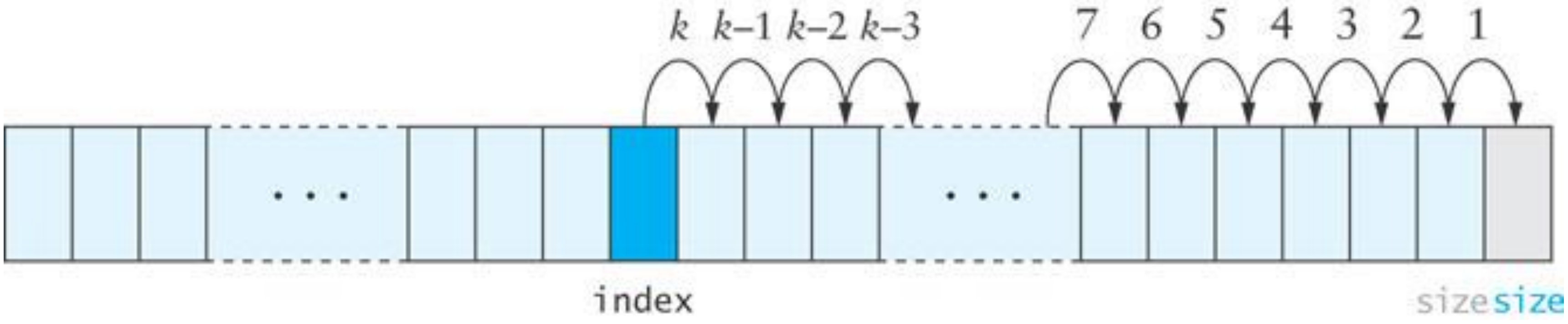
- If `size` is less than capacity, then to append a new item
  1. insert the new item at the position indicated by the value of `size`
  2. increment the value of `size`
  3. return `true` to indicate successful insertion



The method add(E e) runs in O(?)

Implementing `ArrayList.add(int index,E anEntry)`

- To insert into the middle of the array, the values at the insertion point are shifted over to make room, beginning at the end of the array



Implementing ArrayList.add(index,E)

```
public void add (int index, E anEntry) {

  // check bounds
  if (index < 0 || index > size) {
    throw new ArrayIndexOutOfBoundsException(index);
  }

  // Make sure there is room
  if (size >= capacity) {
    reallocate();
  }

  // shift data
  for (int i = size; i > index; i--) {
    theData[i] = theData[i-1];
  }

  // insert item
  theData[index] = anEntry;
  size++;
}
```

The method add(int index, E e) runs in O(?)

get Method

```
public E get (int index) {
  if (index < 0 || index >= size) {
    throw new
ArrayIndexOutOfBoundsException(index);
  }
  return theData[index];
}
```

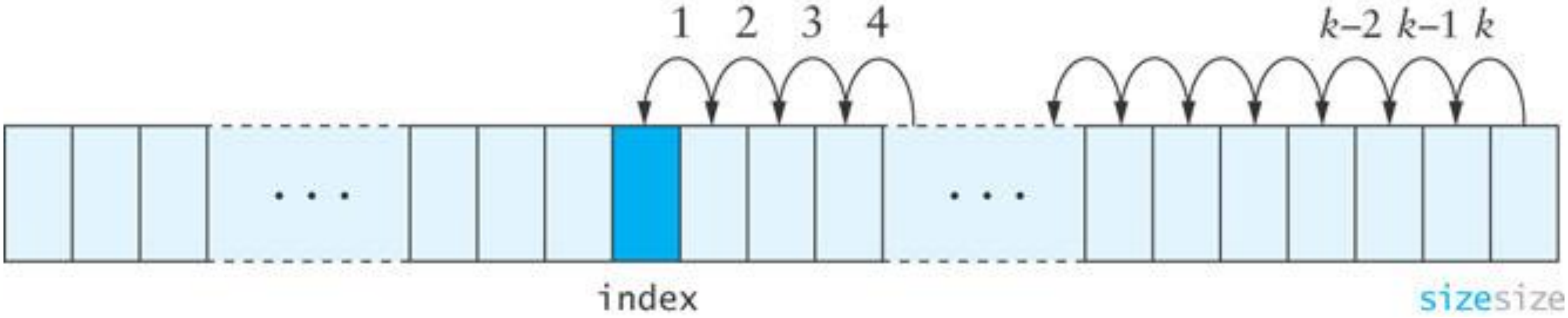
The get() method runs in O(?) time

set Method

```
public E set (int index, E newValue) {
  if (index < 0 || index >= size) {
    throw new
  ArrayIndexOutOfBoundsException(index);
  }
  E oldValue = theData[index];
  theData[index] = newValue;
  return oldValue;
}
```

The method set() runs in O() time

remove Method



- When an item is removed, the items that follow it must be moved forward to close the gap
- Begin with the item closest to the removed element

remove **Method (cont.)**

```
public E remove (int index) {

  if (index < 0 || index >= size) {
    throw new
  ArrayIndexOutOfBoundsException(index);
  }

  E returnValue = theData[index];

  for (int i = index + 1; i < size; i++) {
    theData[i-1] = theData[i];
  }

  size--;
  return returnValue;
}
```

`reallocate` **Method**

- Create a new array that is twice the size of the current array and then copy the contents of the new array

```
private void reallocate () {
  capacity *= 2;
  theData = Arrays.copyOf(theData,
  capacity);
}
```

reallocate Method (cont.)

```
private void reallocate () {
   capacity *= 2;
   theData = Arrays.copyOf(theData,
  capacity);
}
```

The reason for doubling capacity is to spread out the cost of copying;

Performance of `KWArrayList`

- The `set` and `get` methods execute in constant time: O(1)
- Inserting or removing general elements is linear time: O(*n*)
- Adding at the end is (usually) constant time: O(1)
  - With our reallocation technique the average is O(1)
  - The worst case is O(n) because of reallocation

IT 179
8

Single-Linked Lists

For Next Lecture

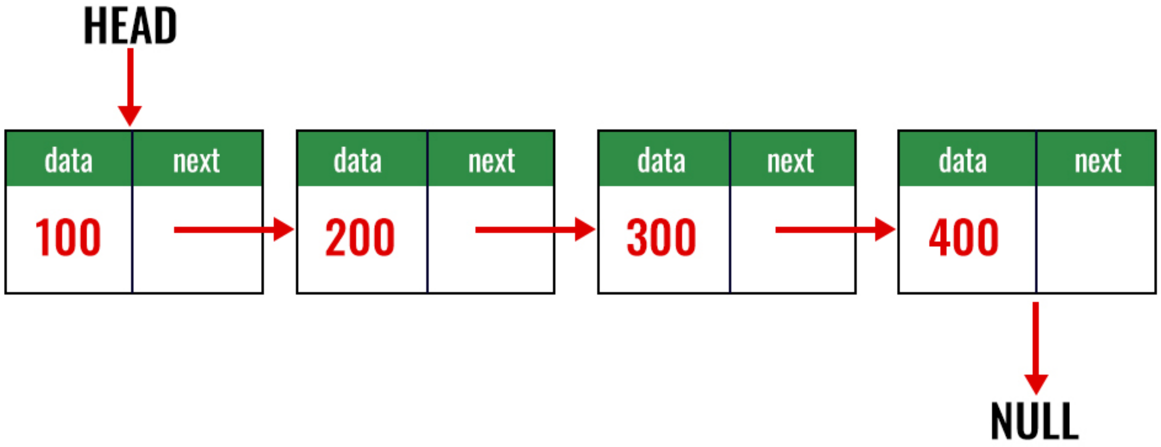
Read Section 2.6
Double-Linked Lists and Circular Lists

Watch video on
Doubly Linked Lists

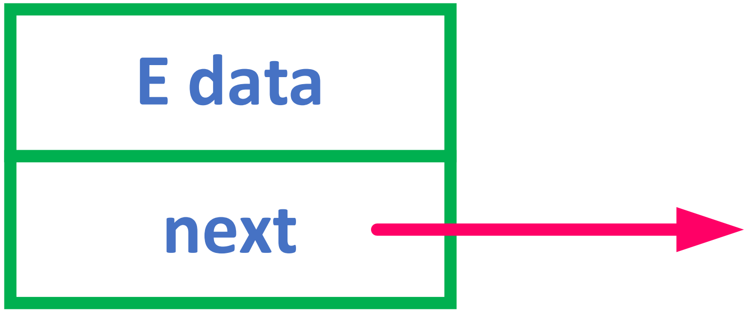
Single-Linked Lists

- A linked list is useful for inserting and removing at arbitrary locations
- The `ArrayList` is limited because its `add` and `remove` methods operate in linear (O(*n*)) time—requiring a loop to shift elements
- A linked list can add and remove elements at a known location in O(1) time
- In a linked list, instead of an index, each element is **linked to the following element**

Example of Single Linked Lists



A List Node



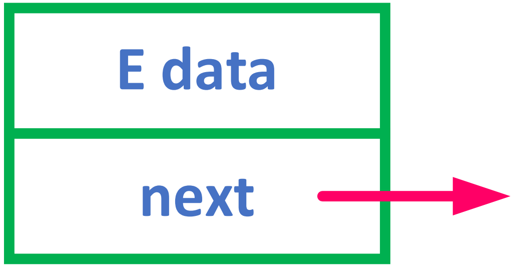
- A node can contain:
  - a data item
  - one or more links
- A link is a **reference** to a list node
- In our structure, the node contains:
  - a data field named `data` of type `E`
  - a reference to the next node, named `next`

Note

To simplify the explanation:

- □ I will **not** use nested classes.
- □ Also, the fields `data` and `next` will not be declared as private.

A List Node



```
public class Node<E>
{
    public E data;
    public Node<E> next;

    /**
     * Creates a new node with a null next field
     *
     * @param dataItem The data stored
     */
    Node(E dataItem)
    {
        this.data = dataItem;
        next = null;
    }

    /**
     * Creates a new node that references another node
     *
     * @param dataItem The data stored
     * @param nodeRef  The node referenced by new node
     */
    Node(E dataItem, Node<E> nodeRef)
    {
        data = dataItem;
        next = nodeRef;
    }
```

A Single-Linked List Class

- Generally, we do not have individual references to each node.
- A **`ISUSingleLinkedList`** object has a data field **`head`**, the *list head*, which references the first list node

```
public class ISUSingleLinkedList<E> {
  private Node<E> head = null;
  private int size = 0;
  ...
}
```

The ISUSingleLinkedList Class

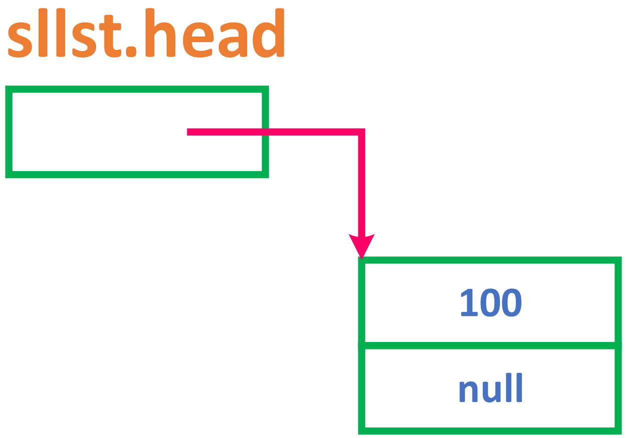
```
public class ISUSingleLinkedList<E>
{

    private Node<E> head = null;
    private int size = 0;


    public void addFirst(E item)
    {
        Node<E> temp = new Node<E>(item, head);
        head = temp;
        size++;
    }
```

Adding one element

```
public static void main(String[] args)
    {
        ISUSingleLinkedList<Integer> sllst = new ISUSingleLinkedList<>();
        sllst.addFirst(100);
    }
```



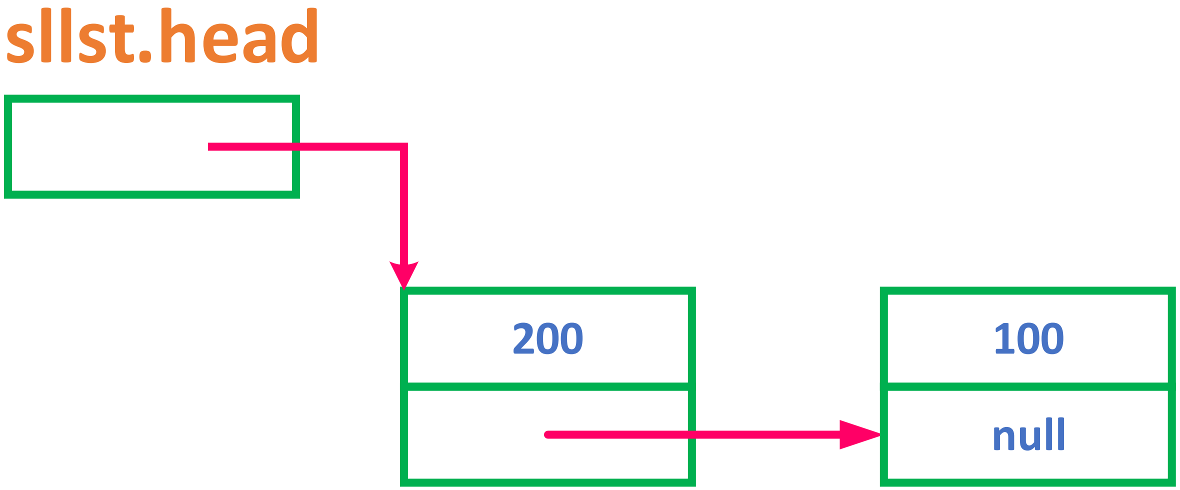
sllst.head

100

null

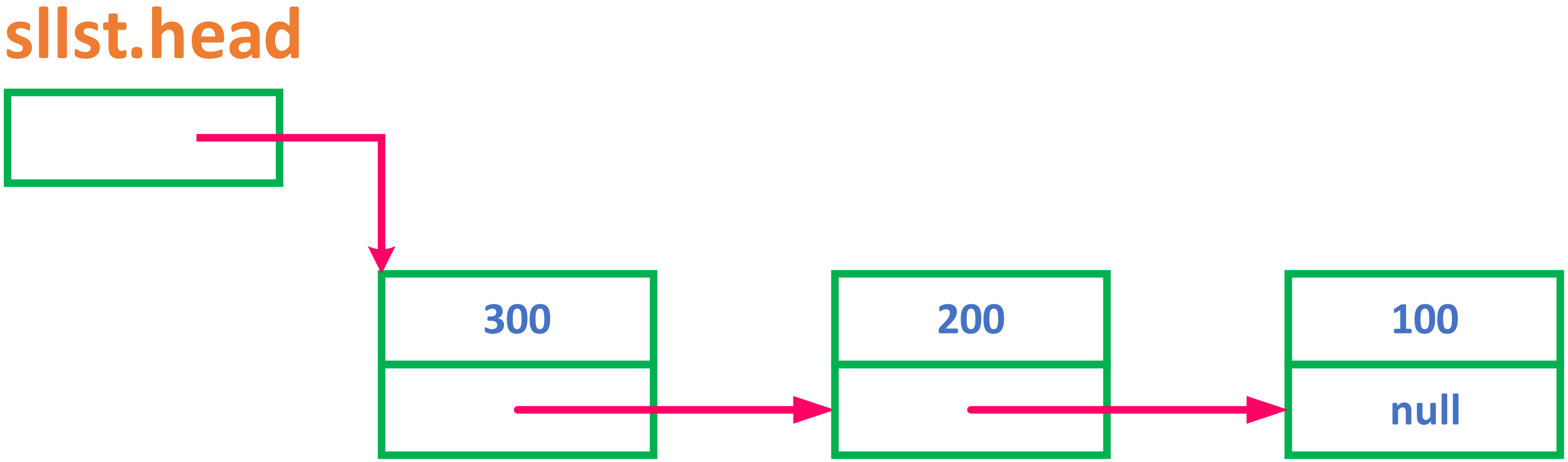
Adding a second element

```
public static void main(String[] args){
        ISUSingleLinkedList<Integer> sllst = new ISUSingleLinkedList<>();
        sllst.addFirst(100);
        sllst.addFirst(200);
    }
```



Adding a third element

```
public static void main(String[] args)    {
        ISUSingleLinkedList<Integer> sllst = new ISUSingleLinkedList<>();
        sllst.addFirst(100);
        sllst.addFirst(200);
        sllst.addFirst(300);
    }
```



Traversing a Single-Linked List

```
public class ISUSingleLinkedList<E>
{

    private Node<E> head = null;
    private int size = 0;

    public void displayList()
    {

        Node<E> currNode = head;
        while (currNode != null)
        {
            System.out.print(currNode.data + " ");
            currNode = currNode.next;
        }

    }

    public void addFirst(E item)
    {
        Node<E> temp = new Node<E>(item, head);
        head = temp;
        size++;
    }
```

Implementing `removeFirst()`

```
public E removeFirst () {
  Node<E> temp = head;
  if (head != null) {
    head = head.next;
  }
  if (temp != null) {
    size--;
    return temp.data
  } else {
    return temp;
  }
}
```

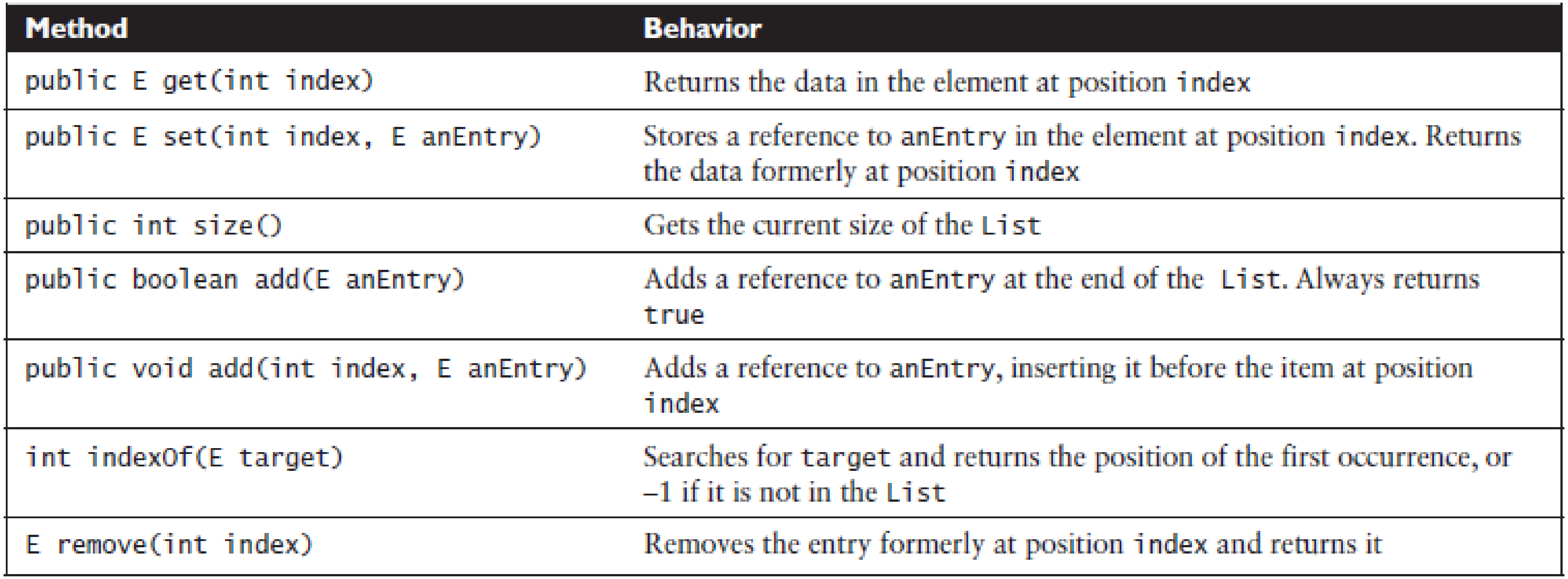
Implementing `removeLast()`

```
public E removeLast()
   {
       Node<E> temp;
       Node<E> prev = head;
       System.out.println("\nRemoving LAST element ...");

       if (prev == null)
           return null;
       if (prev.next == null)
       {
           head = null;
           return prev.data;
       }
       while (prev.next.next != null)
           prev = prev.next;

       temp = prev.next;
       prev.next = null;
       return temp.data;
   }
```

More Methods of `List<E>` Interface in `ISUSingleLinkedList<E>`



Method	Behavior
`public E get(int index)`	Returns the data in the element at position `index`
`public E set(int index, E anEntry)`	Stores a reference to `anEntry` in the element at position `index`. Returns the data formerly at position `index`
`public int size()`	Gets the current size of the `List`
`public boolean add(E anEntry)`	Adds a reference to `anEntry` at the end of the `List`. Always returns `true`
`public void add(int index, E anEntry)`	Adds a reference to `anEntry`, inserting it before the item at position `index`
`int indexOf(E target)`	Searches for `target` and returns the position of the first occurrence, or –1 if it is not in the `List`
`E remove(int index)`	Removes the entry formerly at position `index` and returns it

public E get(int index)

```
public E get (int index) {
 if (index < 0 || index >= size) {
    throw new

IndexOutOfBoundsException(Integer.toString(index));
  }
  Node<E> node = getNode(index);
  return node.data;
}
```

SLList.getNode(int)

- In order to implement methods required by the List interface, we need an additional helper method:

```
private Node<E> getNode(int index) {
  Node<E> node = head;
  for (int i=0; i<index && node != null; i++) {
    node = node.next;
  }
  return node;
}
```

public E set(int index, E newValue)

```
public E set (int index, E newValue) {
  if (index < 0 || index >= size) {
    throw new  I
       IndexOutOfBoundsException(Integer.toString(index));
  }
  Node<E> node = getNode(index);
  E result = node.data;
  node.data = newValue;
  return result;
}
```

public void add(int index, E item)

```
public void add (int index, E item) {
 if (index < 0 || index > size) {
   throw new
     IndexOutOfBoundsException(Integer.toString(index));
 }
 if (index == 0) {
   addFirst(item);
 } else {
   Node<E> node = getNode(index-1);
   addAfter(node, item);
 }
}
```

Implementing `addAfter(Node<E>, E)` (cont.)

```
private void addAfter (Node<E> node, E item) {
  Node<E> temp = new Node<E>(item, node.next);
  node.next = temp;
  size++;
}
```

We declare this method `private` since it should not be called from outside the class. Later we will see how this method is used to implement the public add methods

or, more simply ...

```
private void addAfter (Node<E> node, E item) {
  node.next = new Node<E>(item, node.next);
  size++;
}
```

public boolean add(E item)

- To add an item to the end of the list

```
public boolean add(E item) {
   add(size, item);
   return true;
  }
```

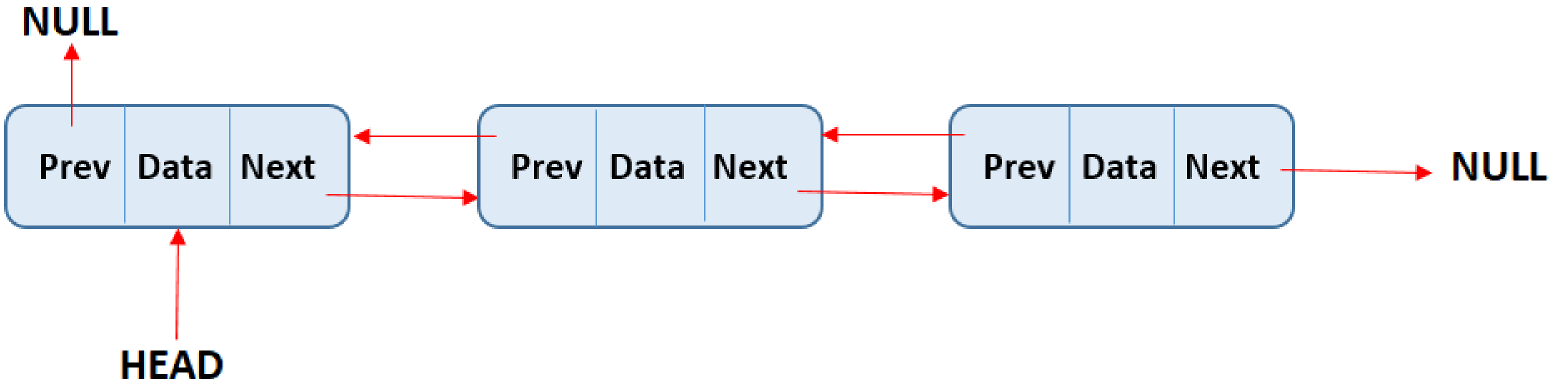
IT 179
9

Double-Linked Lists

Double-Linked Lists

- Limitations of a singly-linked list include:
  - Insertion at the front is O(1); insertion at other positions is O(*n*)
  - Insertion is convenient only after a referenced node
  - Removing a node requires a reference to previous node
  - We can traverse list only in the forward direction
- We can overcome these limitations:
  - Add a reference in each node to the previous node, creating a ***double-linked list***

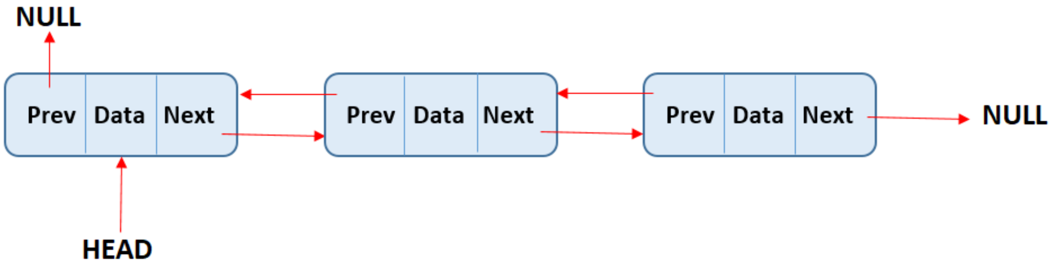
Double-Linked Lists



Node<E> **Class**

```
private static class Node<E> {
  private E data;
  private Node<E> next = null;
  private Node<E> prev = null;

  private Node(E dataItem) {
    data = dataItem;
  }
}
```



A Double-Linked Class

- `head` (a reference to the first list `Node`)
- `tail` (a reference to the last list `Node`)
- `size`

- Insertion at either end is O(1); insertion elsewhere is still O(*n*)