Data Structures

Lists

CS284

Structure of this week's classes

Lists

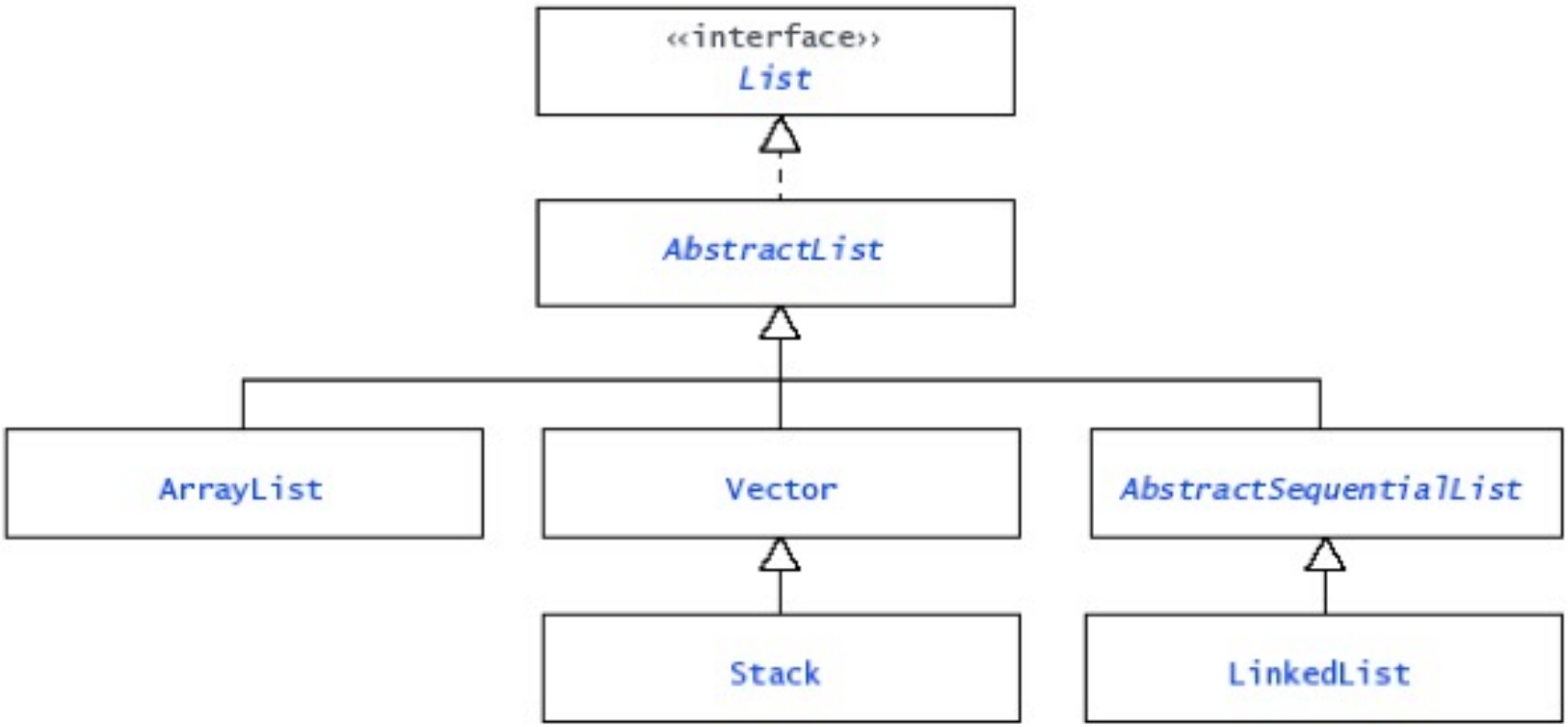
Implementing Lists as Arrays

Implementing Lists as Single-Linked Lists

List

- Sequence of elements with possible duplicates
  - Hence there is first, second, . . ., last element
- Operations:
  - Construct a new list
  - Add an element (end, beginning, index)
  - Remove an element
  - Find an element in the list
  - Check whether the list is empty or not
  - Iterate over its elements
  - . . .

`java.util.List` interface `List<E>`



Sample Methods

`boolean add(E e)`	Append element to the end of this list
`void add(int index, E el)`	Insert element at the specified position in this list
`E get(int index)`	Returns the element at the specified position in this list
`boolean isEmpty()`	Returns true if this list contains no elements
`Iterator<E> iterator()`	Returns an iterator over the elements in this list
`E remove(int index)`	Removes the element at the specified position in this list
`E set(int index, E element)`	Replaces the element at the specified position in this list with the specified element

Two general-purpose implementations

- `ArrayList`: Implementation of lists in terms of arrays
  - Simplest class that implements the List interface
  - Arrays have a fixed size (will require creating new array in order to support adding elements)
  - Constant time access to elements
  - Removal is linear
  - Insertion is linear
- `LinkedList`: Implementation of lists in terms of linked-lists
  - Linked lists may grow and shrink
  - Linear time access
  - Linear time insertion and removal (except if previous element supplied, then constant)

`ArrayList` implementation

- The list is stored in an array, which is a private member of the ArrayList (you cannot access it)
- This array has a capacity
- When the number of elements in the list exceeds the capacity, the internal array is replaced by a bigger one

Let's look at an example

Creating a simple List

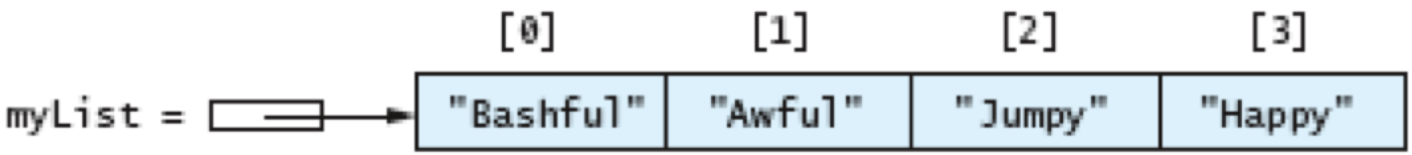
```
// Declare a List ``object'' whose elements
// will reference String objects
List<String> myList= new ArrayList<String>();

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

Creating a simple List

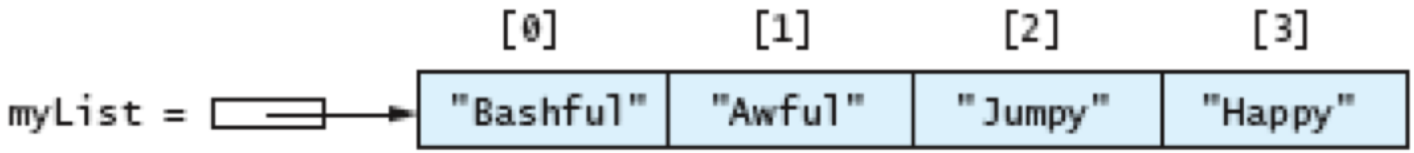
```
// Declare a List ``object'' whose elements
// will reference String objects
List<String> myList= new ArrayList<String>();

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

Creating a simple List

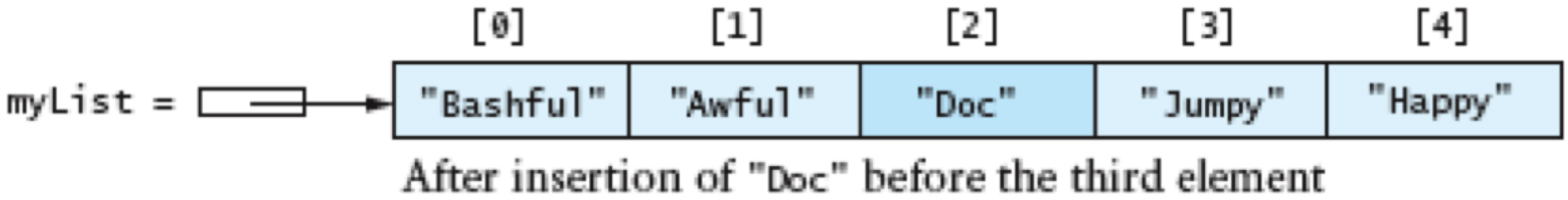
```
// Declare a List ``object'' whose elements
// will reference String objects
List<String> myList= new ArrayList<String>();

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



Adding an element




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



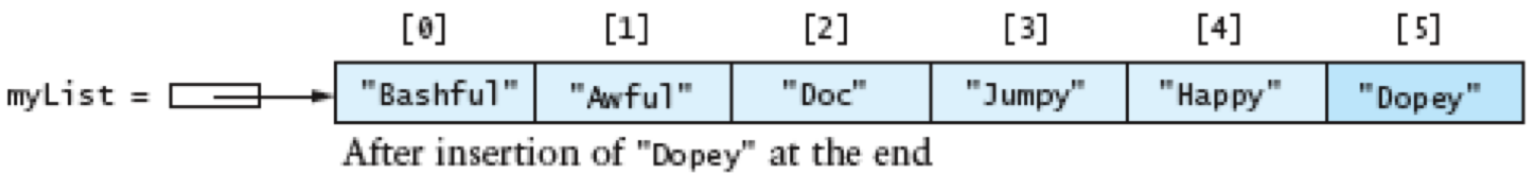
After insertion of "Doc" before the third element

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

Adding an element

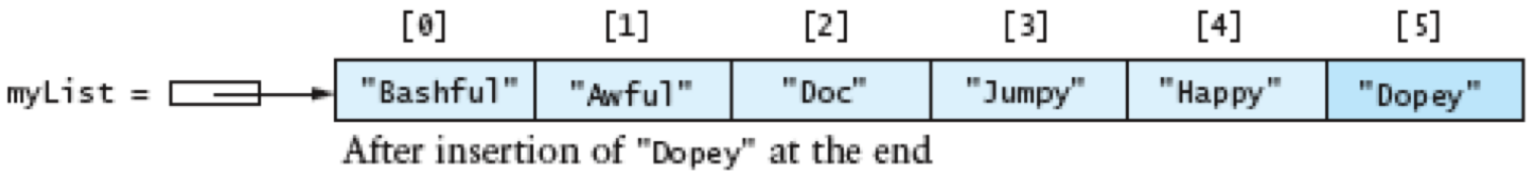
- When no subscript is specified, an element is added at the end of the list:

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



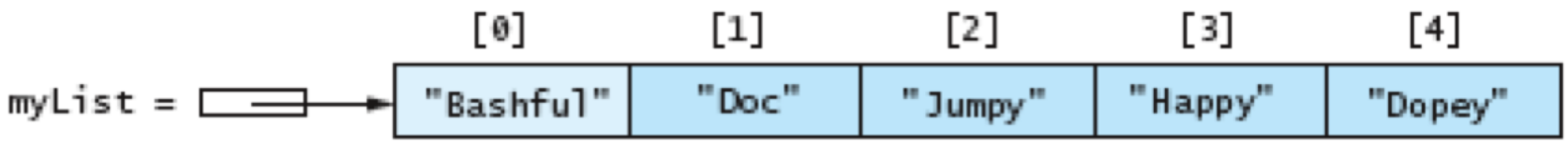
After insertion of "Dopey" at the end

Removing an element



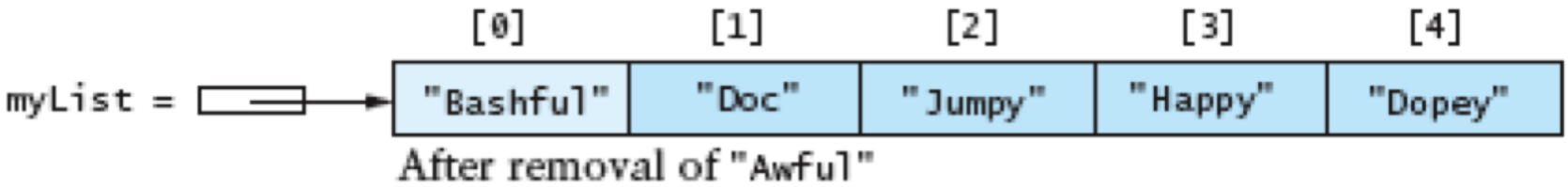
After insertion of "Dopey" at the end

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

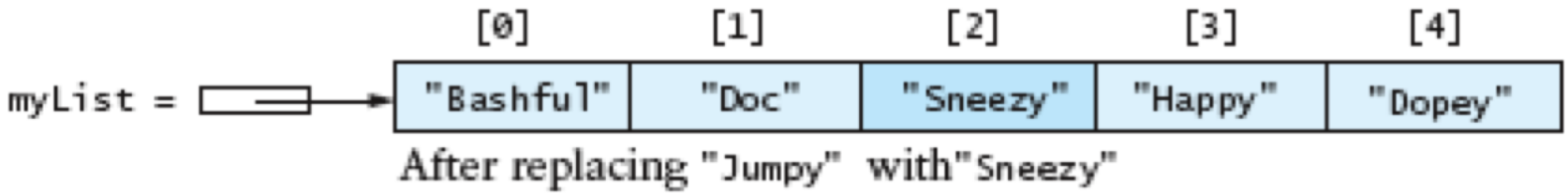


After removal of "Awful"

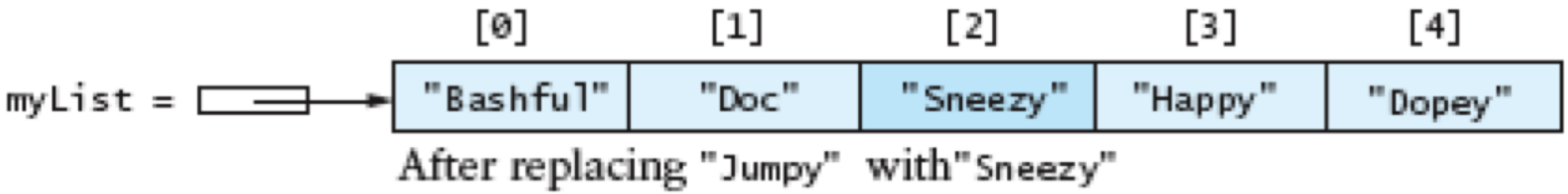
Replacing an element




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



Accesing an element

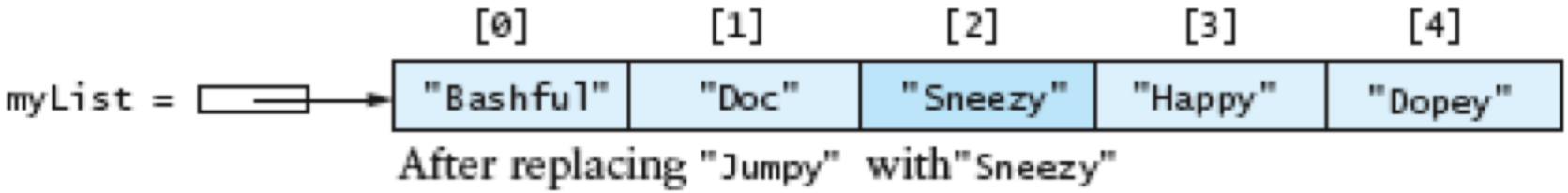


You can access an element using the get() method:

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

The value of dwarf becomes “Sneezy”

Searching




```
myList.indexOf("Sneezy");
// returns 2

myList.indexOf("Jumpy");
// returns -1 (unsuccessful search)
```

Note on use of feature called *generics*

```
List<Integer> myList = new ArrayList<Integer>();
myList.add(new Integer(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 at compile-time
```

Benefits of using generics

- Better type-checking: catch more errors, catch them earlier
- Documents intent
- Avoids the need to downcast from `Object`

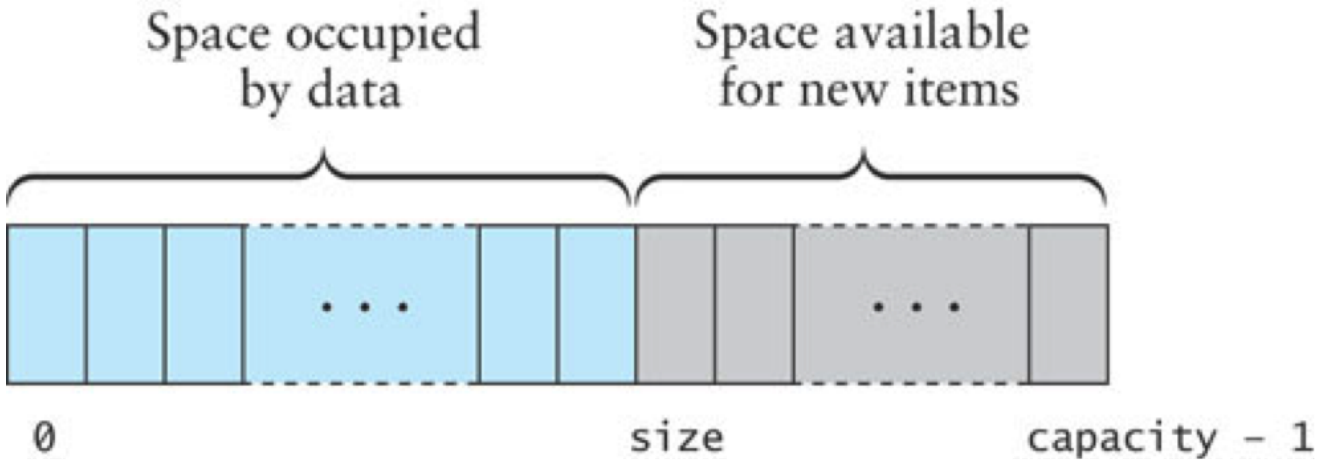
Lists

Implementing Lists as Arrays

Implementing Lists as Single-Linked Lists

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



```java
import java.util.*;

/** This class implements some of the methods of the Java Arra
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];
}
```

KWArrayList Constructor

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

- Set initial capacity

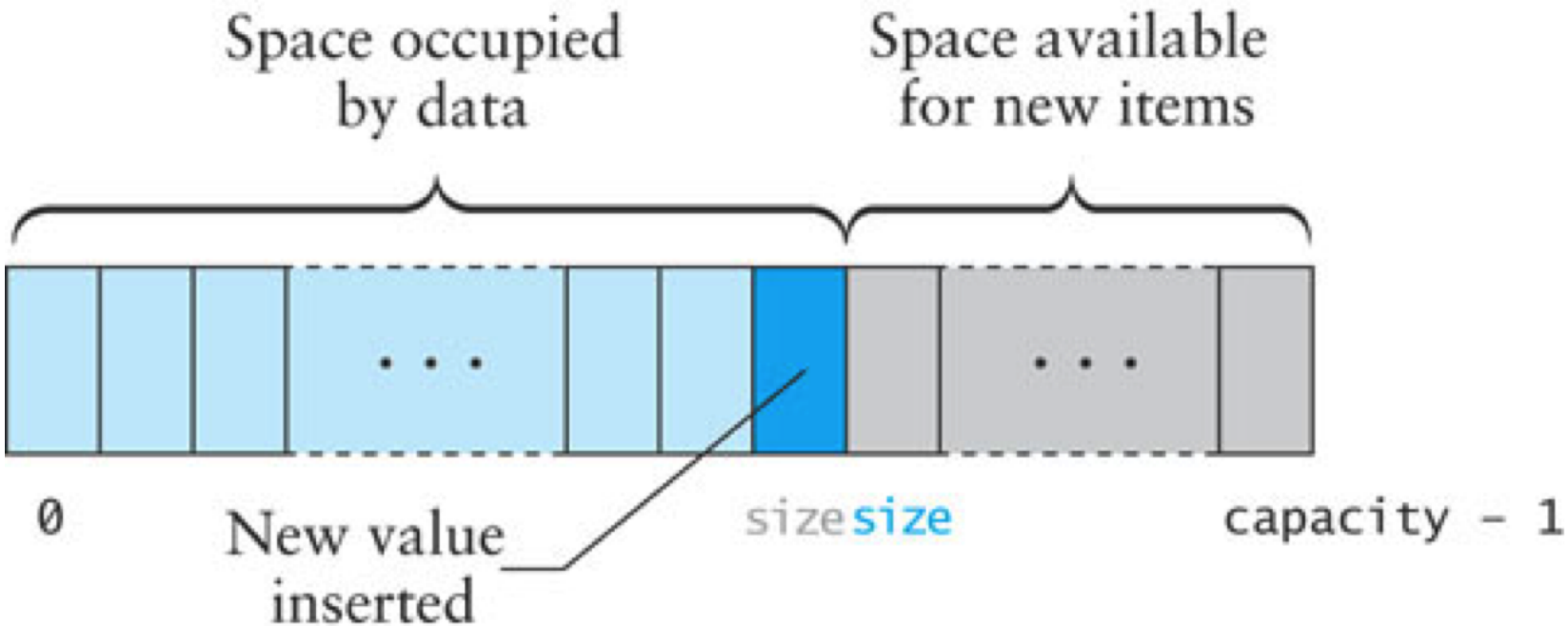
KWArrayList Constructor

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

- Set initial capacity
- Create new array of type `Object` and then cast to array of type `E`

Adding an element at a specified index

1. If size is greater or equal to capacity, then make room
2. insert the new item at the position indicated by the value of size
3. increment the value of size



`boolean add(E anEntry)`

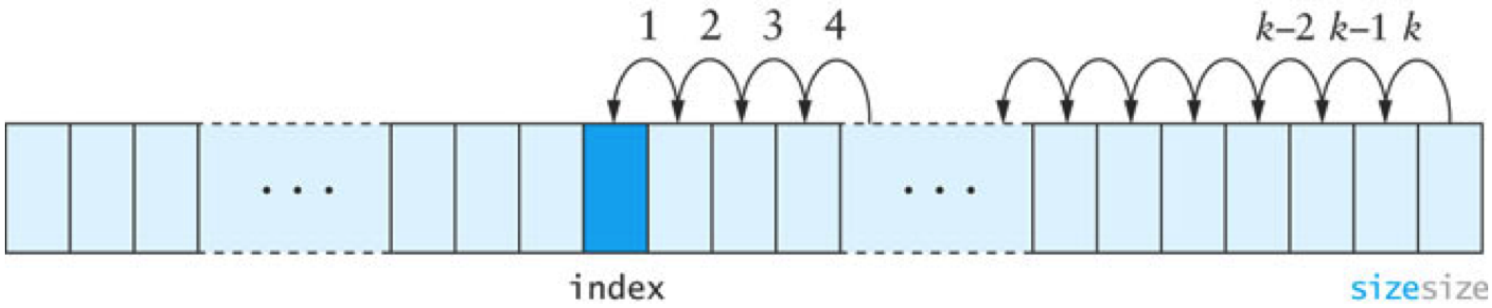
```
public boolean add(E anEntry) {
    if (size == capacity) {
        reallocate();
    }
    theData[size] = anEntry;
    size++;
    return true;
}
```

```java
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++;
}
```

`get` and `set` methods

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

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;
}
```

Removing an element



- 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 and proceed in the indicated order

Removing an element

```
public E remove(int index) {

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

  E returnValue = theData[index];

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

  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

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

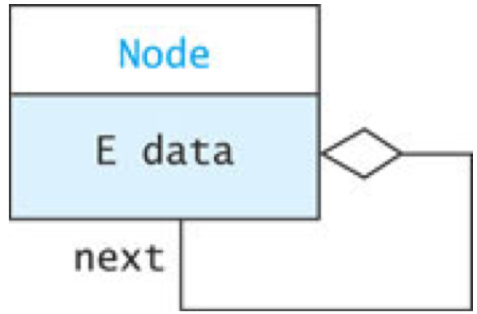
```
private void reallocate() {
  capacity *= 2;
  theData = Arrays.copyOf(theData, capacity);
}
```

- Copies the `theData`, padding with nulls so the copy has the specified length

Lists

Implementing Lists as Arrays

Implementing Lists as Single-Linked Lists




```
public class LinkedList<E> {
  private static class Node {
    private E data;
    private Node next; // defaults to null

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

- A static nested class is behaviorally a top-level class that has been nested in another top-level class for packaging convenience

An additional constructor for `Node`

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

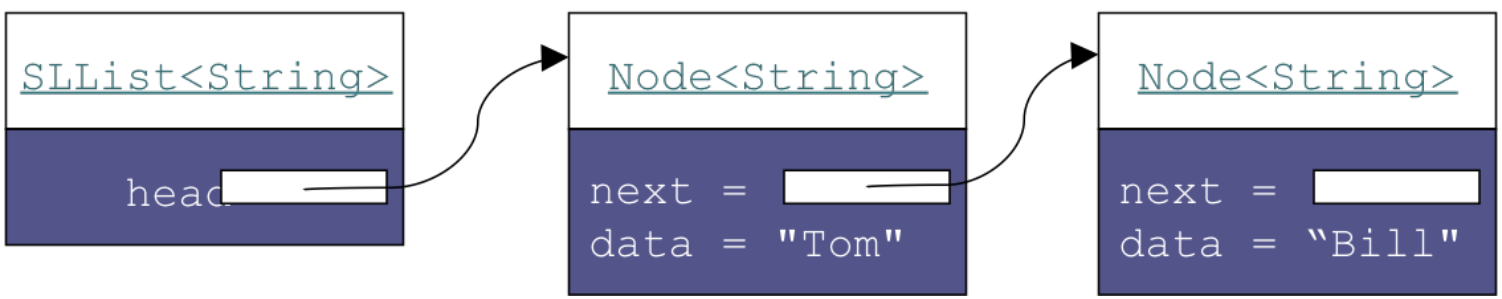
Connecting Nodes

```
Node<String> tom = new Node<String>("Tom");
Node<String> bill = new Node<String>("Bill");
Node<String> harry = new Node<String>("Harry");
Node<String> sam = new Node<String>("Sam");

tom.next = bill;
bill.next = harry;
harry.next = sam;
```

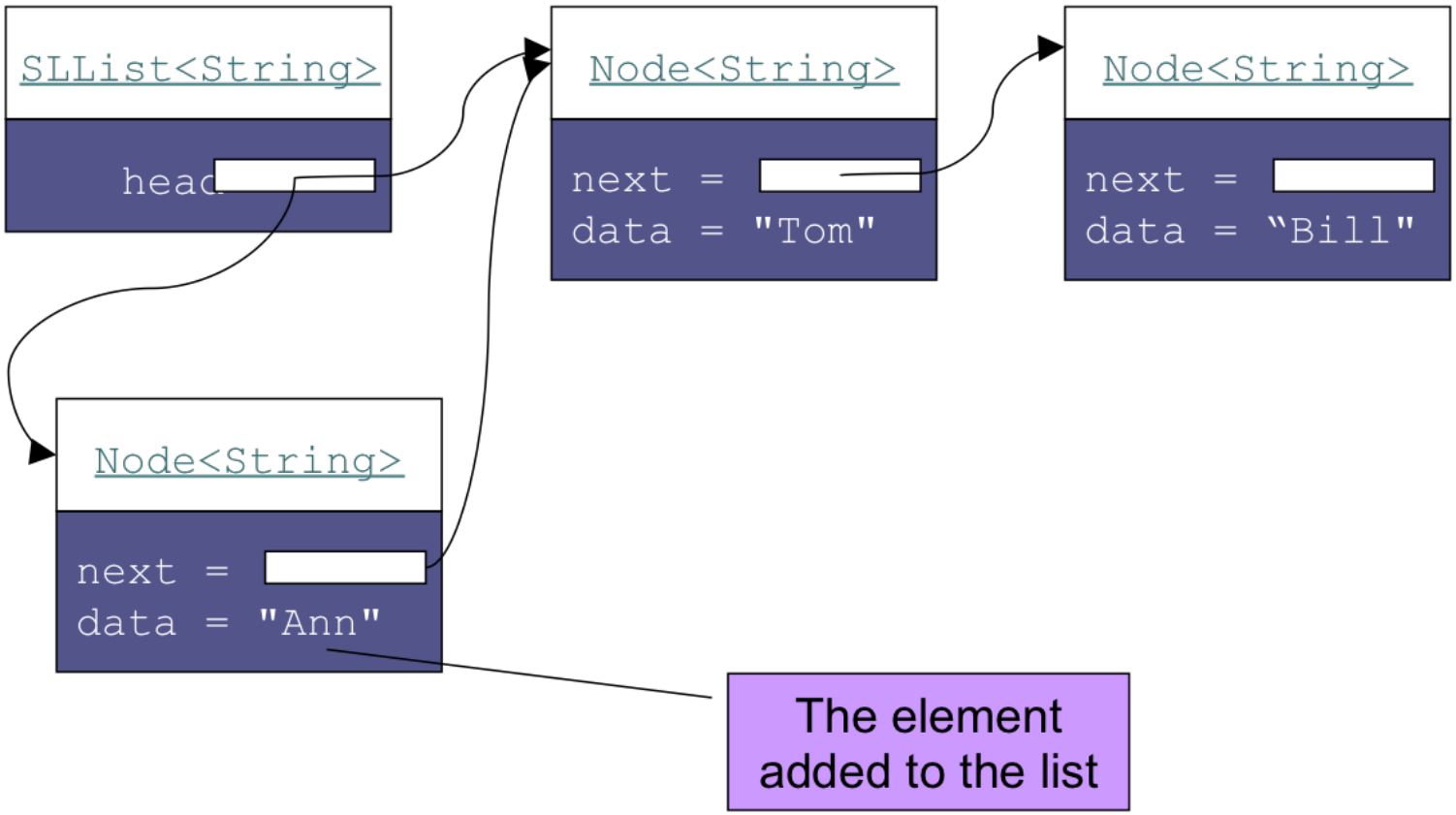
A Single Linked-List Class

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

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



SLList.addFirst(E item)



SLList.addFirst(E item)

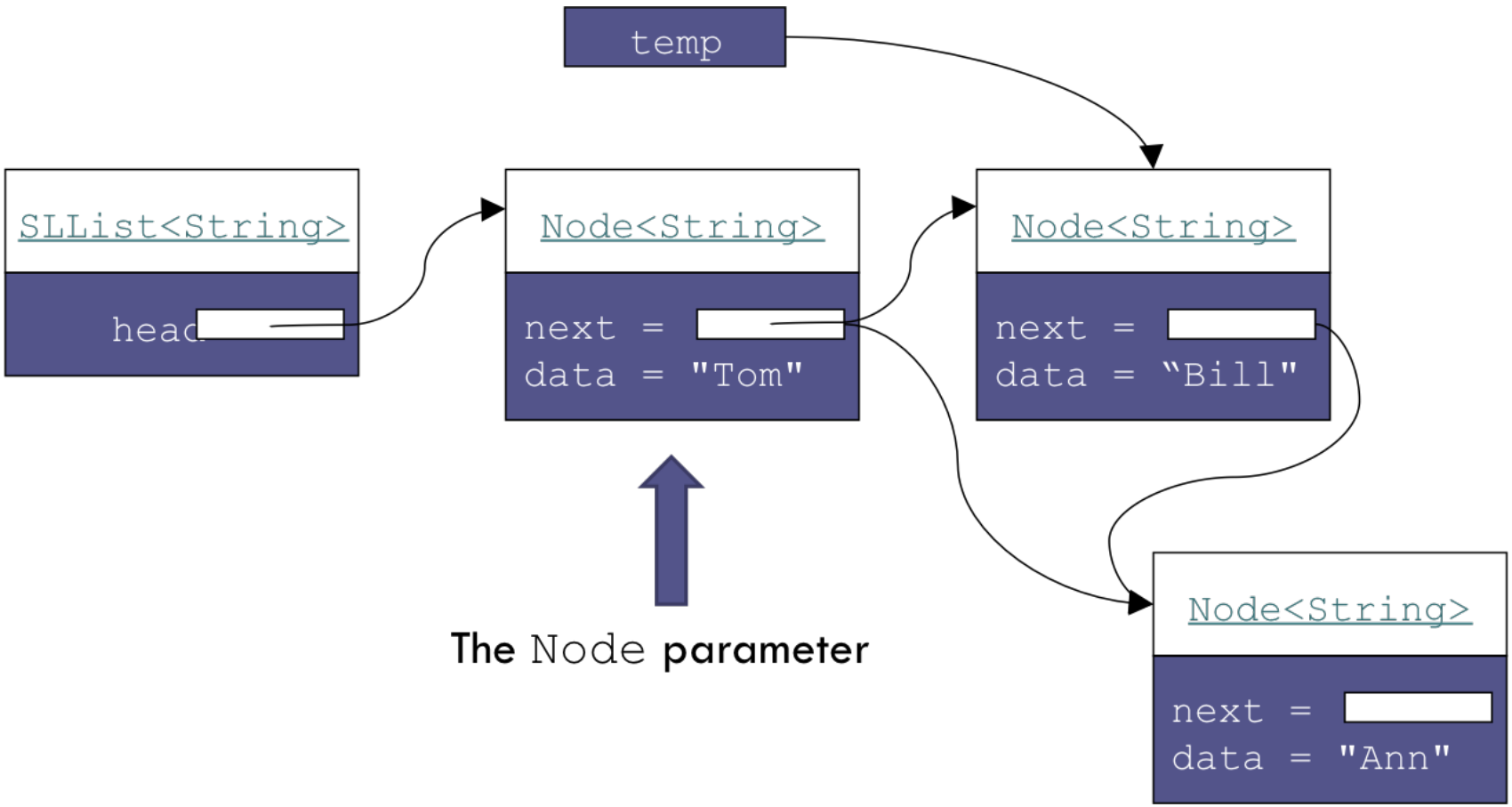
```
private void addFirst(E item) {
  head = new Node(item, head);
  size++;
}
```

- Does this work if `head` is **`null`**?
- 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.

addAfter(Node node, E item)

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

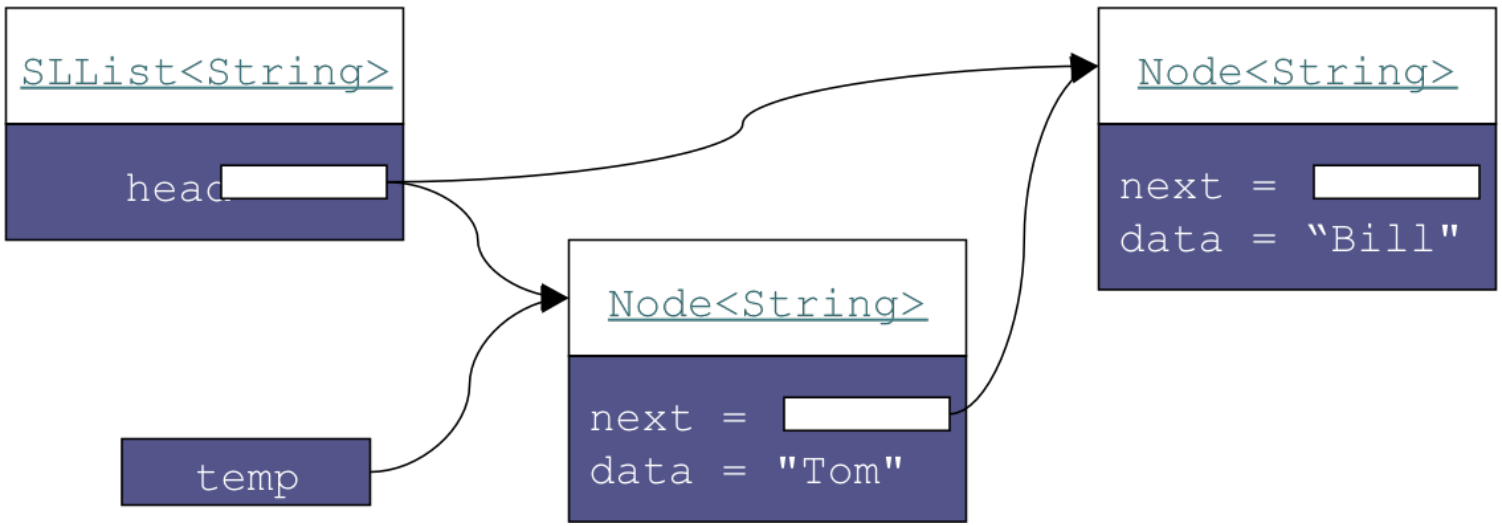
Implementing `removeAfter(Node node)`



Implementing `removeAfter(Node node)`

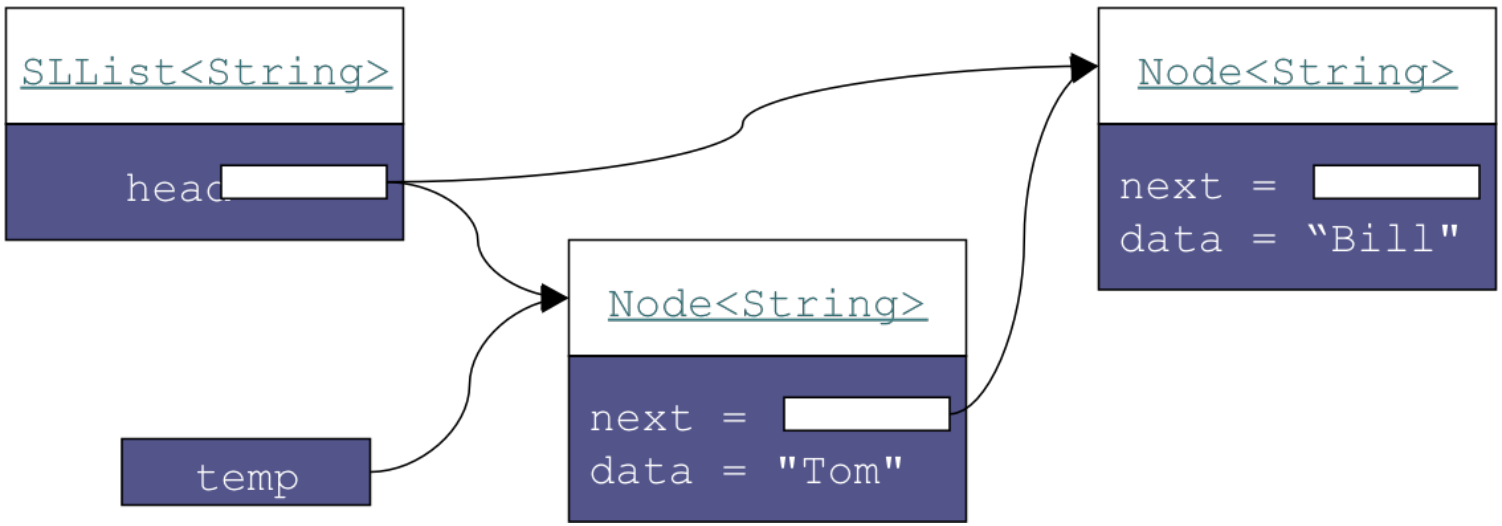
```
private E removeAfter(Node node) {
  Node temp = node.next;
  if (temp != null) {
    node.next = temp.next;
    size--;
    return temp.data;
  } else {
    return null;
  }
}
```

RemoveFirst



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

Traversing a linked-list



Traversing a linked-list

```
@Override  // Optional, but recommended
public String toString() {
    StringBuilder sb = new StringBuilder("[");
    Node p = head;
    if (p != null) {
        while (p.next != null) {
            sb.append(p.data.toString());
            sb.append(" ==> ");
            p = p.next;
        }
        sb.append(p.data.toString());
    }
    sb.append("]");
    return sb.toString();
}
```

- `StringBuilder` objects are like `String` objects, except that they can be modified

public E get(int index)

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

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

public E set(**int** index, E anEntry)

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

public boolean add(E item)

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