

MAHARISHI UNIVERSITY of MANAGEMENT

Engaging the Managing Intelligence of Nature

Computer Science Department

CS390 Fundamental Programming

Practices (FPP)

Professor Paul Corazza and

Professor Ankhtuya Ochirbat

Lecture 8:

The List Data Structure

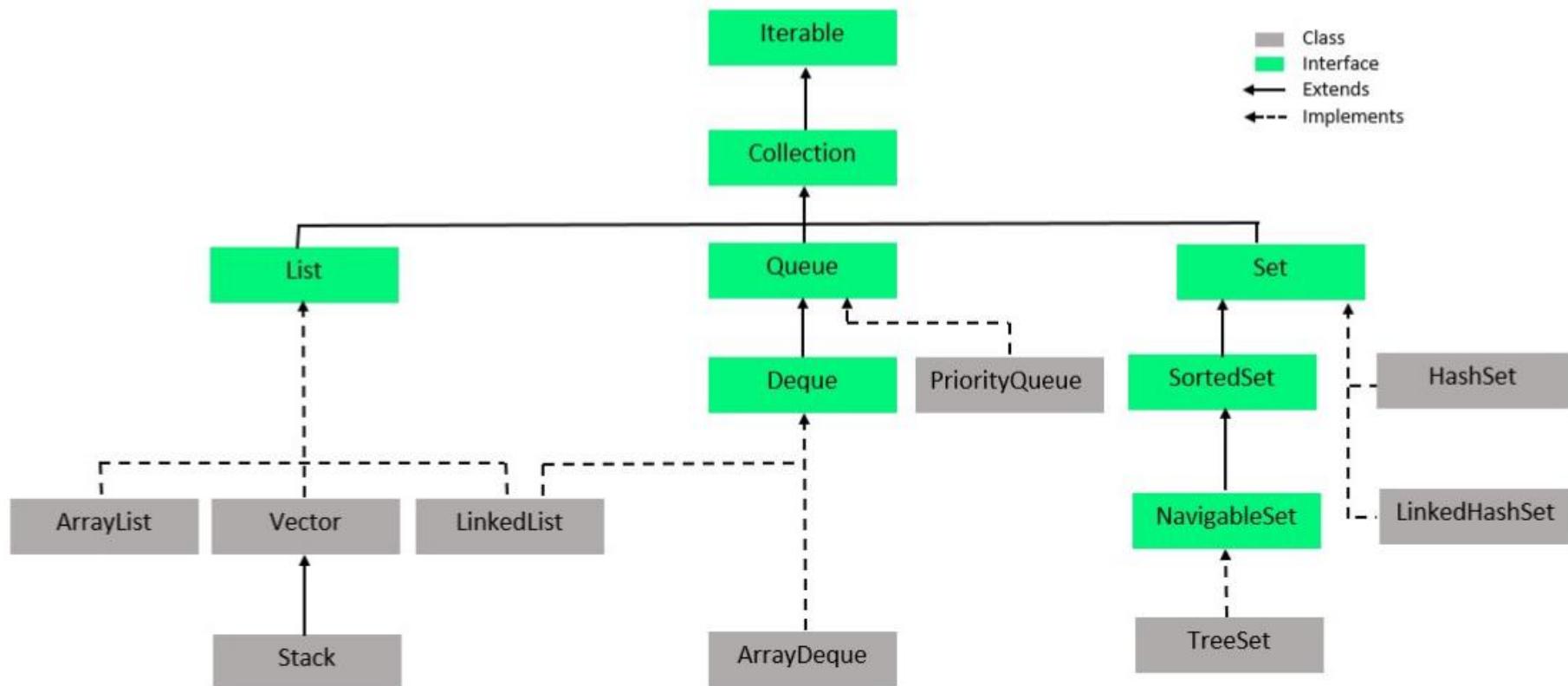
Wholeness of the Lesson

The List ADT is one of the most general data types, capable of supporting most needs for storing a collection of objects in memory. Different implementations of this data type provide optimizations for different operations – such as insert, delete, find – that are typically supported by Lists. Lists give expression to the natural tendency of pure intelligence to express itself through a sequential unfoldment.

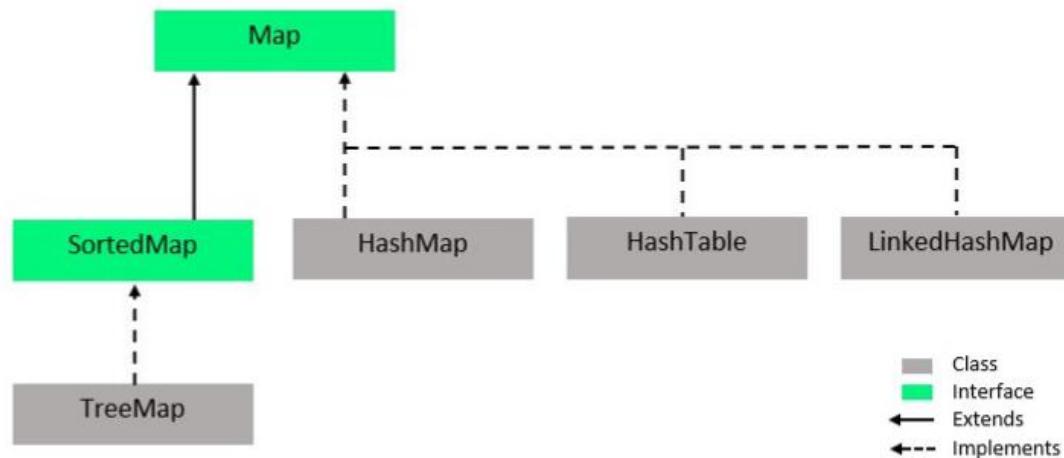
Outline of Topics

- List ADT, Array Lists and including sort and search
- Linked Lists – singly linked, headers, doubly linked, circular linked
- Lists before and after jse5.0
- The List interface, the AbstractList class, and Iterator
- Collections.sort, Collections.binarySearch, and RandomAccess
- Four ways of iterating through elements in a list
- Comparators

Collection Framework



Collection Framework



Collection interface

Methods	Description
boolean add(object o)	Returns true if the object is added to the specified collection
boolean addAll(collection c)	Returns true if the entire object in the collection is added to the specified collection.
Void clear()	Removes all elements from the specified collection.
boolean equals(object o)	Returns true if the specified object matches with the object in the collection.
boolean isEmpty()	Returns true if the collection is empty.
boolean remove(object o)	Returns true, if the specified element is present in the collection and removes the object from the collection.
boolean removeAll(collection c)	Returns true, if all the elements in collection c is removed from the specified collection.
boolean retainAll(collection c)	Returns true, if all the elements in collection c is retained in the specified collection.
int size()	Returns the number of elements in the collection.
object[] toArray()	Returns an array containing all of the elements in the collection.
object[] toArray(object[] a)	Returns an array of object if the array contains all the elements in the specified collection.

List interface

Method	Description
void add(int index, object o)	Adds the element, o in the specified index of the list.
boolean addAll(int index, collection c)	Adds all the elements of collection, c in the specified index of the list.
Eget(int index)	Returns the element available in the specified index of the list.
int indexsOf(object o)	Returns the index of object o in the list.if there are more than one occurrence of object o, the method returns the index of the first occurrence.if the object o is not available in the list, the method returns -1.
int lastindexOf(object O)	Returns the last index of the object o in the list.if the object o is not available in the list, the method returns -1.
Eremove(int index)	Removes the element at the specified index of the list.

The LIST Abstract Data Type

- "List" is an *abstract data type* (ADT) – consisting of a sequence of objects and operations on them.
- Typical operations:

<i>find(Object o)</i>	returns position of first occurrence
<i>findKth(int pos)</i>	returns element based on index
<i>insert(Object o, int pos)</i>	inserts object into specified position
<i>remove(Object o)</i>	removes object
<i>printList()</i>	outputs all elements
<i>makeEmpty()</i>	empties the List

- Other operations are sometimes included, like "contains".
- Can be implemented in more than one way. *ArrayList* is one such implementation.

ArrayList: A Growable Array

- *Arrays Are Very Efficient* Arrays are data structures that provide "random access" to elements – to find the *i*th entry, there is no need to traverse the elements prior to the *i*th in order to locate the *i*th entry.
- *Arrays Inconvenient Because of Fixed Length.* Arrays are inconvenient sometimes because it is necessary to commit to a fixed array size before adding elements. If the number of elements then exceeds the array size, a new larger array must be created to accommodate the new elements, and old elements have to be copied into the new array. There are similar problems involved in removing elements and in inserting elements into a specified position.
- *ArrayList.* A convenient data structure that saves the explicit effort of recopying. Here, all the work required to copy over elements into a new array for insert, remove, and adding operations is encapsulated in the class.
- Example: MyStringList in prog3_3 (lab 3) and lesson8.demo.mystringlist

java.util.ArrayList<E>

```
+ArrayList()  
+add(o: E): void  
+add(index: int, o: E): void  
+clear(): void  
+contains(o: Object): boolean  
+get(index:int): E  
+indexOf(o: Object): int  
+isEmpty(): boolean  
+lastIndexOf(o: Object): int  
+remove(o: Object): boolean  
+size(): int  
+remove(index: int): boolean  
+set(index: int, o: E): E
```

ArrayList

import java.util.ArrayList;

Name	Use
add(item)	adds item to the end of the list
add(index,item)	adds item at index – shifts items up->
set(index,item)	changes item at spot (z[index]=item)
get(index)	returns the item at index (return z[index])
size()	returns the # of items in the list
remove(int index)	removes item at index from the list and returns it

Array Operations Can Be Included in An ArrayList's Set of Methods

- We consider two operations: *sorting* and *searching a sorted array*
- There are many sorting algorithms; Java provides a sorting routine as part of its API. We will consider a simple one for illustration.
- *MinSort* uses the following approach to perform sorting an array A of integers.
 - Start by creating a new array B that will hold the final sorted values
 - Find the minimum value in A, remove it from A, and place it in position 0 in B.
 - Place the minimum value of the remaining elements of A in position 1 in array B.
 - Continue placing the minimum value of the remaining elements of A in the next available position in B until A is empty.

In-Place MinSort for Arrays

In-Place MinSort. MinSort can be implemented without an auxiliary array. This is done by performing a swap after each min value is found. Here is the code:

```
//arr is given as input
int[] arr;
public void sort(){
    if(arr == null || arr.length <=1) return;
    int len = arr.length;
    for(int i = 0; i < len; ++i){
        //find position of min value from arr[i] to arr[len-1]
        int nextMinPos = minpos(i, len-1);

        //place this min value at position i
        swap(i, nextMinPos);
    }
}

//Swaps values arr[i], arr[j]
void swap(int i, int j){
    int temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
}

//Returns pos of min value from
//positions i to j
int minpos(int i, int j){
    int pos = i; int min = arr[i];
    for(int k = i + 1; k <= j; ++k) {
        if(arr[k] < min) {
            pos = k; min = arr[k];
        }
    }
    return pos;
}
```

Exercise 8.1

Include a version of in-place MinSort in MyStringList. Since Strings will be compared instead of ints, you will need to use the compareTo method of the String class.

```
public class MyStringList {  
    private final int INITIAL_LENGTH = 2;  
    private String[] strArray;  
    private int size;  
  
    public MyStringList() {  
        strArray = new String[INITIAL_LENGTH];  
        size = 0;  
    }  
  
    public void minSort() {  
        //implement  
    }  
}
```

Solution

```
public void minSort(){
    if(strArray == null || size<=1) return;
    for(int i = 0; i < size; ++i){
        int nextMinPos = minpos(i,size-1);
        swap(i,nextMinPos);
    }
}
void swap(int i, int j){
    String temp = strArray[i];
    strArray[i] = strArray[j];
    strArray[j] = temp;
}

//find minimum of arr between the indices bottom and top
public int minpos(int bottom, int top){
    String min = strArray[bottom];
    int index = bottom;
    for(int i = bottom+1; i <= top; ++i){
        if(strArray[i].compareTo(min)<0){
            min = strArray[i];
            index = i;
        }
    }
    //return location of min, not the min itself
    return index;
}
```

Searching a Sorted Array with Binary Search

- If an array `arr` of integers is already sorted, we can search for a given integer `testVal` in a very efficient way (a recursive implementation is given in Lab 7-3)

Let `mid = arr[arr.length/2]` (the value in the middle position of the array).

- If `testVal == mid`, return true
- Else if `testVal < mid`, search for `testVal` in the left half of the array
- Else if `testVal > mid`, search for `testVal` in the right half of the array

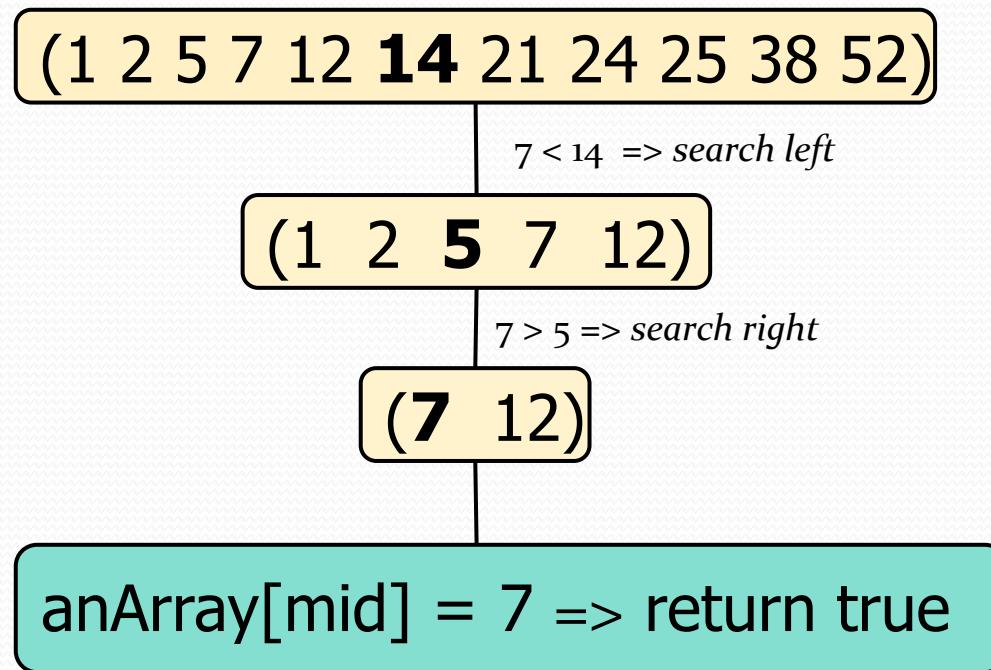
Online demo: <http://www.cs.armstrong.edu/liang/animation/web/BinarySearch.html>

Here is code for performing iterative binary search on a sorted array.

```
public static boolean binSearchIt(int[] arr, int x) {  
    if(arr == null || arr.length == 0) return false;  
    int lower = 0;  
    int upper = arr.length - 1;  
    while(true) {  
        if(lower > upper) return false;  
        int mid = (lower + upper)/2;  
        if(x == arr[mid]) return true;  
        if(x < arr[mid]) { //search left  
            upper = mid - 1; //lower is not changed  
        } else { //x > arr[mid], search right  
            lower = mid + 1; //upper is not changed  
        }  
    }  
}
```

Example

Search key val = 7



Example

Search key val = 20

(1 2 5 7 **12** 14 21 24 25 38)

$20 > 12 \Rightarrow \text{search right}$

(14 21 **24** 25 38)

$20 < 24 \Rightarrow \text{search left}$

(**14** 21)

$20 > 14 \Rightarrow \text{search right}$

(**21**)

$20 < 21 \Rightarrow \text{search left}$

lower >upper => return false

Searching a Sorted List with Binary Search

- The strategy of repeatedly cutting the size of the search domain by a factor of 2 makes this algorithm highly efficient. It does **NOT** work if the array is not already sorted.
- Lab prog8_1: Implement a version of binary search in MyStringList.

Inefficiencies of ArrayList insert, add, remove Operations

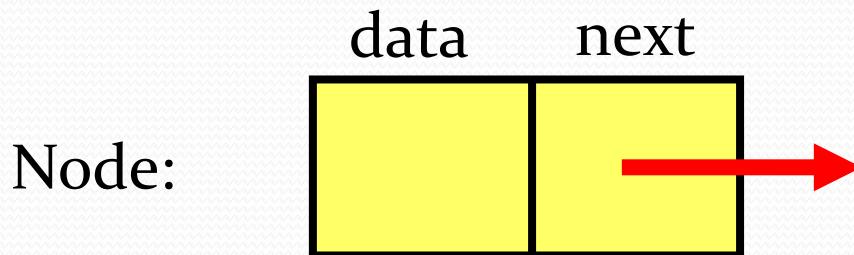
- If, in using an ArrayList, the operations remove, insert, and add are used predominantly, performance is not optimal because of repeated resizing and other steps that require array copying. For such purposes, another implementation of List is better.

Outline of Topics

- List ADT, Array Lists and including sort and search
- **Linked Lists – singly linked, headers, doubly linked, circular linked**
- Generic types for lists before and after jse5.0
- The List interface, the AbstractList class, and Iterator
- Collections.sort, Collections.binarySearch, and RandomAccess
- Four ways of Iterating Through Elements in a List
- Comparators

LinkedList Implementation of LIST: Concept of a NODE

A LinkedList consists of Nodes. Nodes are structures made up of a data field and a link to the next Node (which may be null).



Node Data Structure

```
public class Node {  
    String data;  
    Node next;  
}
```

Exercise 8.2

In a `main` method, try adding and removing data to form various-length linked lists, starting from a single `Node`. For instance, the following code first creates a length-1 list, then a length-2 list:

```
//Single-node list
Node startNode = new Node();
startNode.data = "A";
System.out.println(startNode);

//Two-node list
Node next = new Node();
next.data = "B";
startNode.next = next;
System.out.println(startNode);
```

Output:

```
A
A B
```

Continue using this logic to produce a three-node list, to remove B from the list, and to insert a new node X into position 1 in the list.

LinkedList Implementation of LIST

- *The Need:* Improve performance of *insert*, *remove*, *add*, and avoid the cost of resizing incurred by the array implementation. [See demo `lesson8.node.full.MainSoln`]
- *Operations* – Placed *inside* the Linked List instead of being executed from an external class (as in examples above).
 - *search* and *add* both require traversing the Nodes via links, starting at the first node either till value is found or till next node is null (in latter case, if *searching*, return "not found"; if *adding*, create new node and make it the new next node)
 - *insert* requires traversing the nodes to locate position and adjusting links
 - *remove* requires doing a *search*, and when the object is found, the *previous* object has to be located so that it can be linked to the *next* object

Linked List Skeleton

```
class StringLinkedList {  
    class Node { .. }  
    void addFirst(String) { .. }  
    void addLast(String) { .. }  
    boolean search(String) { .. }  
    int size() { .. }  
    boolean remove(String) { .. }  
    void insert(String s, int pos) { .. }  
    Boolean remove(int pos) {.. }  
}
```

Implementing the *addFirst* Operation

- *addFirst* must create a new node containing input string and setting its next to be the current startNode.

```
public void addFirst(String s) {  
    Node newNode = new Node();  
    newNode.data = s;  
    newNode.next = startNode;  
    startNode = newNode;  
}
```

Implementing the *addLast* Operation

- *addLast* requires traversing nodes via links till next node is null; the next node is then instantiated and populated with the input value

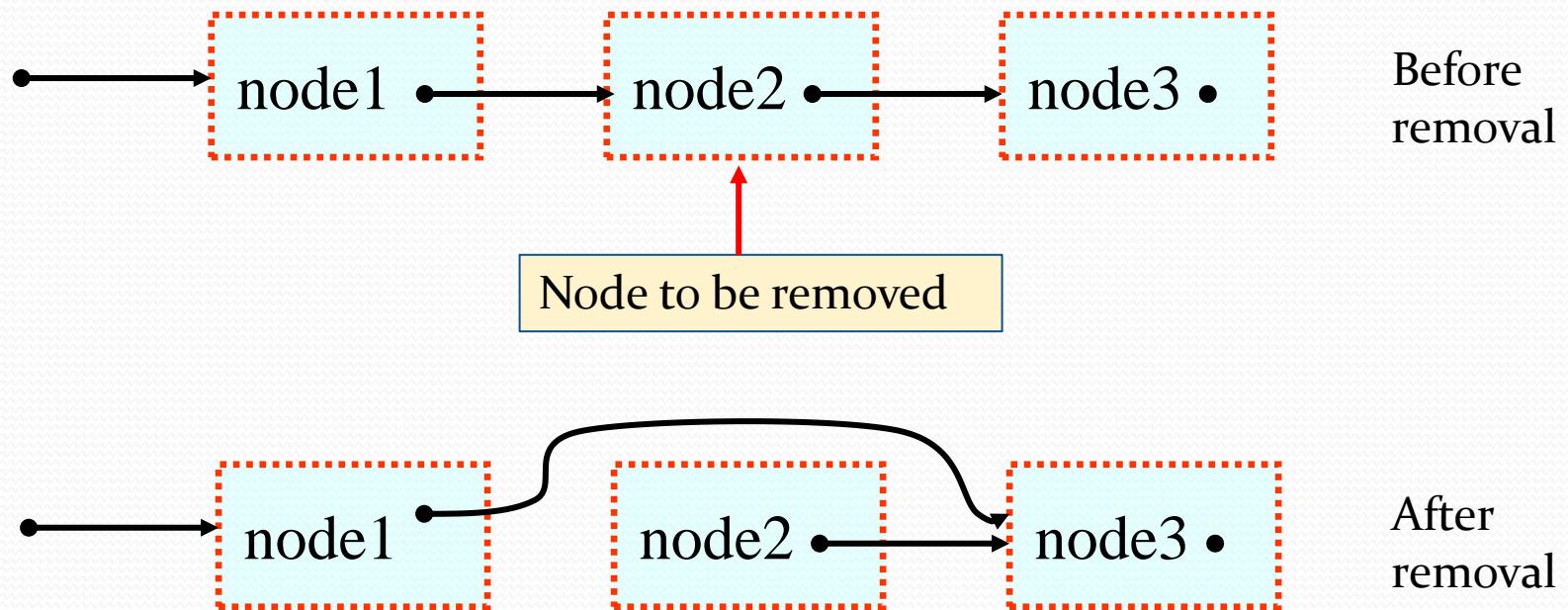
```
public void addLast(String s) {  
    Node newNode = new Node();  
    newNode.data = s;  
    //if startNode == null, set startNode to be newNode  
    if(startNode == null) {  
        startNode = newNode;  
    }  
    else { //find last non-null node  
        Node last = startNode;  
        while(last.next != null) {  
            last = last.next;  
        }  
        //now last is the last non-null node  
        last.next = newNode;  
    }  
}
```

Implementing *search* Operation

- *search* requires traversing the nodes via links, starting at the `startNode`, either till value is found (and "true" is returned) or till next node is null (in which case "false" is returned)

```
boolean search(String s) {  
    if(s == null) return false;  
    Node temp = startNode;  
    while(temp != null) {  
        String t = temp.data;  
        if(s.equals(t)) {  
            return true;  
        }  
        temp = temp.next;  
    }  
    return false;  
}
```

Implementing *remove* Operation

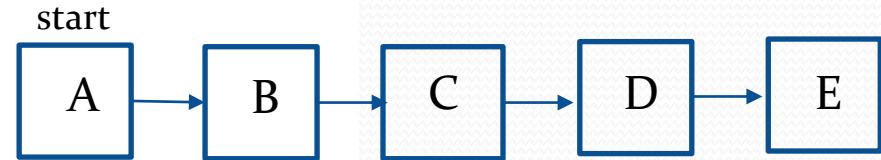


Implementing *removeNode* Operation

An important step in `removeNode` is being able to find the previous node.

- `removeNode` method could invoke a routine to go back to the beginning and locate the previous node
- `removeNode` method could maintain a reference to previous node (best)

```
/** Removes node containing input string s and returns true, if found
 * Otherwise, returns false.
 */
boolean removeNode(String s) {
    if(s == null || startNode == null) return true;
    if(startNode.data.equals(s)){
        startNode = startNode.next;
        return true;
    }
    Node previous = startNode;
    Node next = startNode.next;
    while(next != null) {
        if(s.equals(next.data)) {
            previous.next = next.next;
            return true;
        }
        previous = next;
        next = next.next;
    }
    //String s not found, return false
    return false;
}
```



Implementing the *size* operation.

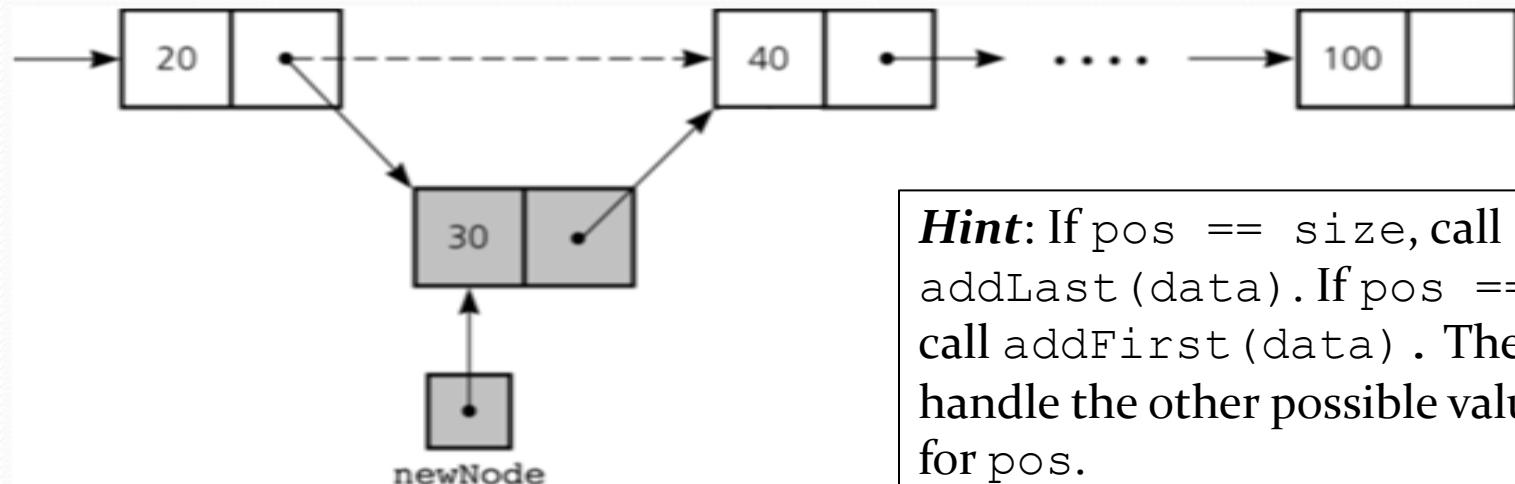
Involves traversing nodes following links and keeping count of how many nodes have been visited. Another way is to maintain size as an instance variable, updating after adds, insertions, and removals.

```
/** size = the number of non-null nodes */
int size() {
    if(startNode == null) return 0;
    Node temp = startNode;
    int count = 0;
    while(temp != null) {
        count++;
        temp = temp.next;
    }
    return count;
}
```

Exercise 8.3: Implementing the insert Operation

- insert requires traversing the nodes to locate position and adjusting links
- inserts a new node containing data so that its position in the list is now pos

```
void insert(String data, int pos)
```



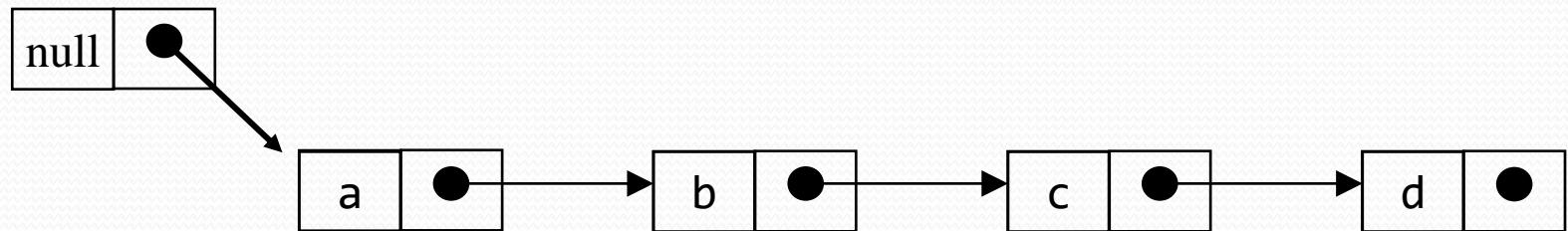
Hint: If `pos == size`, call `addLast(data)`. If `pos == 0`, call `addFirst(data)`. Then handle the other possible values for `pos`.

Solution

```
void insert(String s, int pos) {  
    int size = size();  
    if(pos < 0 || pos > size) {  
        throw new IllegalArgumentException(  
            "Illegal position for new node");  
    }  
    if(pos == size) addLast(s);  
    else if(pos == 0) addFirst(s);  
    else {  
        Node n = new Node();  
        n.data = s;  
        //startNode will not be null here  
        Node previous = startNode;  
        for(int i = 0; i < pos - 1; ++i) {  
            previous = previous.next;  
        }  
  
        //insert n just after previous  
        n.next = previous.next;  
        previous.next = n;  
    }  
}
```

Linked Lists with Headers

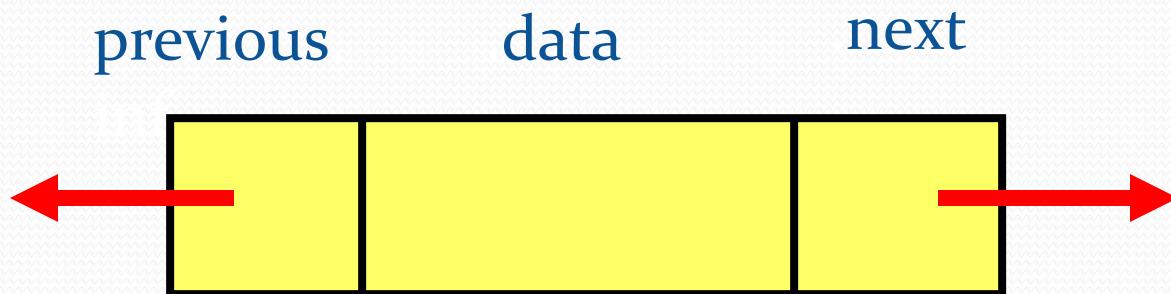
header



- A header is a Node that contains no data, can never be removed, and has a link to first Node. This is how a singly linked list is usually implemented.
- Including a header makes the operations `addFirst` and `remove` easier to implement because special cases for `startNode` do not need to be handled
- Demo: `lesson8.singlylinkedwithheader`

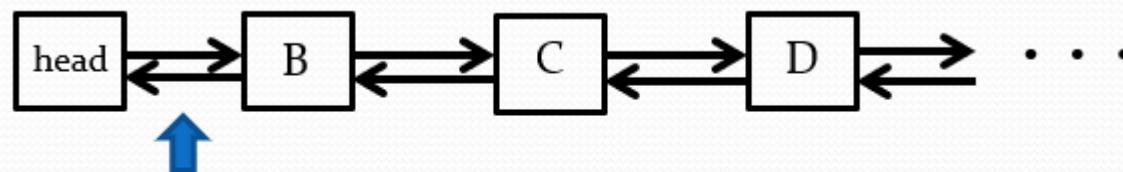
Doubly Linked List

- Each node contains three fields: data stored in the node, a link to the previous node, and a link to the next node.



Doubly Linked List with Header

```
public class MyStringLinkedList {  
    Node header;  
    MyStringLinkedList(){  
        header = new Node(null);  
    }  
    //adds to the front of the list  
    public void addFirst(String item){  
        Node n = new Node(item);  
        //place new node after header and  
        //establish links from new node to  
        //surrounding nodes  
        n.next = header.next;  
        n.previous = header;  
  
        //establish links from surrounding  
        //nodes to the new node  
        if(header.next != null){  
            header.next.previous = n;  
        }  
        header.next = n;  
    }  
}
```



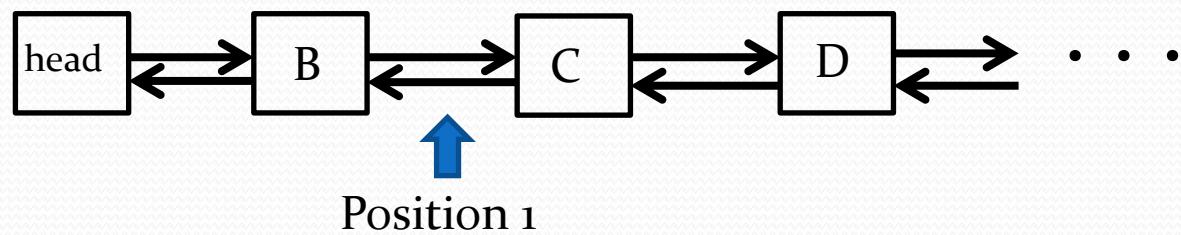
Doubly linked lists with header are the most commonly used type of linked list. They are more efficient than singly linked lists, but implementation requires more care.

lesson8.doublylinked.MyStringLinkedList

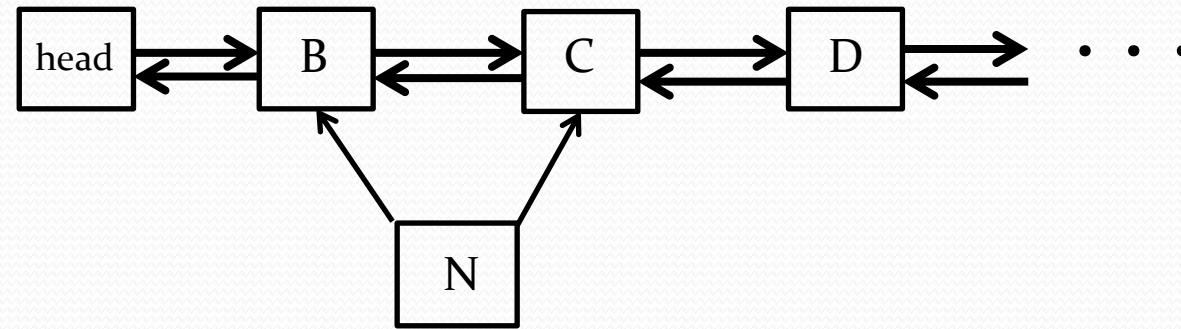
Question: How to implement the insert and remove operations?

Inserting at a Specified Position

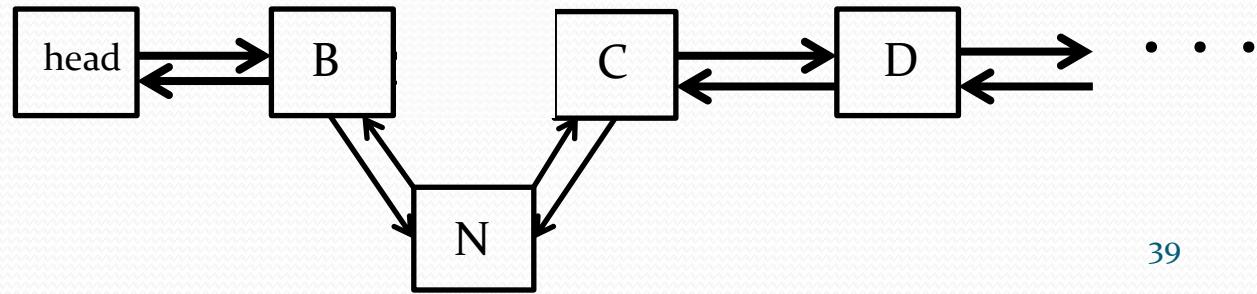
1. Locate position to insert



2. Create new node N and create links next and previous to existing nodes

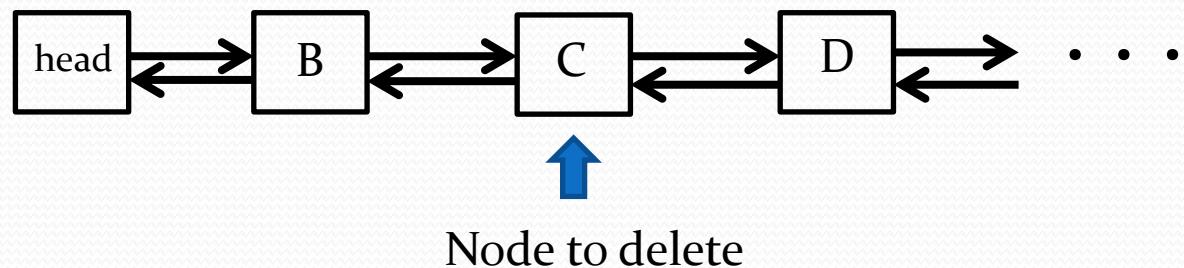


3. Break links between existing nodes and set next and previous links into N

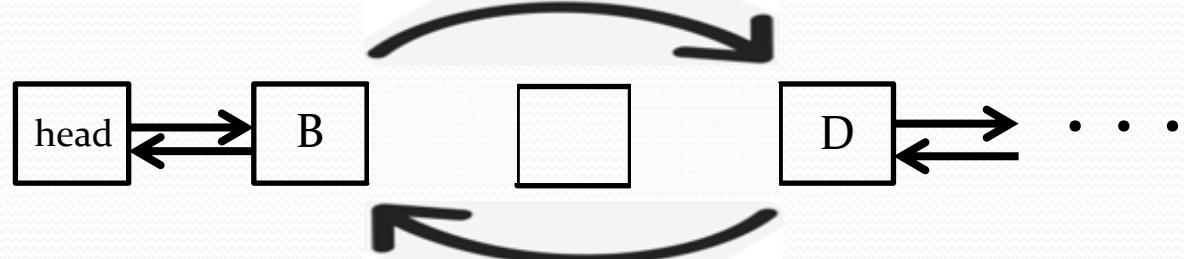


Deleting a Node

1. Locate position of node C to be deleted.



2. Break links into and out of node C. Join C.previous and C.next with links.



Example: removeLast

```
public void removeLast( ) {  
    //if list is empty, return  
    if(header.next == null) return;  
  
    Node current = header;  
    //traverse the list until current.next is  
    //null - then remove current  
    while(current.next != null) {  
        current = current.next;  
    }  
    //now current.next == null, so remove current  
    Node previous = current.previous;  
    //previous is not null since current is not header  
    previous.next = null;  
    current.previous = null;  
}
```

Example: Implementing search

```
public boolean search(String s) {  
    Node next = header.next;  
    while(next != null && !next.value.equals(s)) {  
        next = next.next;  
    }  
    //either next == null or next.value is s  
    if(next == null) return false;  
    else { //next.value.equals(s)  
        return true;  
    }  
}
```

Circular Linked Lists

- In a circular linked list, the last element has a link to the first
- If a header is used, the last element links to the header
- If the `LinkedList` is doubly linked, and has a header, `header.previous` points to the last element
- Making a doubly linked list circular cuts the search time for the operations `insert (Object o, int pos)` and `findKth` in half.

Sorting and Binary Search in Linked Lists

- Sorting routines that rely on the indices of elements in the list are inefficient when used on linked lists, since locating elements by index requires a full traversal of the list. The fastest sorting routines do rely on indices. Therefore, the most efficient way of sorting a linked list is to copy elements from the linked list to an array, perform fast sorting on the array, and then copy elements back to the linked list.
- There is a version of MinSort that does not rely on indices, but it is relatively inefficient so it is not used in practice.
- Binary Search on sorted linked lists requires repeated access to the "middle" index, and this requirement reduces performance. Binary Search on linked lists is no more efficient than scanning the list.

Main Point

The List ADT captures the abstract notion of a “list”; it specifies certain operations that any kind of list should support (for example, *find*, *findKth*, *insert*, *remove*), without specifying the details of implementation.

Different concrete implementations of this abstract data type (such as Array Lists and Linked Lists) meet the contract of the List ADT using different implementation strategies. Likewise, pure awareness is an abstraction of individual awareness; each individual provides a specific, concrete realization of pure consciousness.

Outline of Topics

- List ADT, Array Lists and including sort and search
- Linked Lists – singly linked, headers, doubly linked, circular linked
- **Generic types for lists before and after jse5.0**
- The List interface, the AbstractList class, and Iterator
- Collections.sort,
Collections.binarySearch, and
RandomAccess
- **Four ways of Iterating Through Elements in a List**
- Comparators

Genericising the Objects Stored in a List

- One difficulty with our examples of Lists – MyStringList and MyStringLinkedList – is that they don't work if the objects we wish to store are not Strings.
- *Unsatisfactory Solution:* Rewrite the List code for each type as the need arises. E.g. MyEmployeeList, MyIntegerList, MyAccountList...
- *A Better Solution:* Could create a List that stores elements of type Object.

Example: MyObjectList

```
public class MyObjectList {  
    private final int INITIAL_LENGTH = 4;  
    private Object[] objArray;  
    private int size;  
  
    public MyObjectList() {  
        objArray = new Object[INITIAL_LENGTH];  
        size = 0;  
    }  
  
    public void add(Object ob) {  
        if(size == objArray.length) resize();  
        objArray[size++] = ob;  
    }  
    . . .  
}  
  
//USAGE  
MyObjectList list = new MyObjectList();  
list.add("Bob");  
list.add("Sally");  
String name = (String)list.get(1); //downcast necessary
```

Example: MyObjectLinkedList

```
public class MyObjectLinkedList {  
    Node header;  
    MyObjectLinkedList(){  
        header = new Node(null);  
    }  
  
    public void addFirst(Object item){  
        Node n = new Node(item);  
        n.next = header.next;  
        n.previous = header;  
        if(header.next != null){  
            header.next.previous = n;  
        }  
        header.next = n;  
    }  
}
```

```
class Node {  
    Object value;  
    Node next;  
    Node previous;  
    Node(Object value){  
        this.value = value;  
    }  
    public String toString() {  
        return value == null ?  
            "null" : value.toString();  
    }  
}
```

```
//USAGE  
MyObjectLinkedList list  
    = new MyObjectLinkedList();  
list.add("Bob");  
list.add("Sally");  
String name = (String)list.get(1);
```

Java's Approach (before jdk 1.5)

- Before j2se5.0, Java provided versions of these two kinds of Lists having implementations similar to the above.
- Java's ArrayList. This is an array-backed list that accepts any type of object, like MyObjectList above.

Usage:

```
ArrayList list = new ArrayList();
list.add("Bob");
list.add("Sally");
```

```
String name = (String)list.get(1);
```

- Java's LinkedList. This is a linked list that accepts any type of object, like MyObjectLinkedList above.

Usage:

```
LinkedList list = new LinkedList();  
list.add("Bob");  
list.add("Sally");  
String name = (String)list.get(1);
```

Parametrized Lists in JSE5.0

- To do away with the downcasting and support compiler type checking, the Java designers created *parametrized lists* in j2se5.0.
- An example of an undesirable aspect of old-style lists (which parametrized lists fix) is the following:

```
List list2 = new ArrayList();
list2.add("mike");
Integer i = (Integer) list2.get(0);
```

Runtime exception because there is no compiler checking of types in a collection.

- Starting with j2se5.0, Lists include a generic parameter. Here are declarations from the Java library:

```
class ArrayList<E> implements List<E> {
    ArrayList<E>() {
        ...
    }
}

class LinkedList<E> implements List<E> {
    LinkedList<E>() {
        ...
    }
}

interface List<E> {
    void add(E ob);
    E get(int pos);
    boolean remove(E ob);
    int size();
    . . .
}
```

Demo : lesson8.generic.list

```
//USAGE
List<String> list = new ArrayList<String>();
list.add("Bob");
list.add("Sally");
String name = list.get(0); //no downcast required

//iterate using for each construct - no downcasting needed
for(String s : list) {
    //do something with s
}

//clumsy runtime exceptions are now replaced by compiler errors
List<Integer> list = new ArrayList<Integer>();
list.add(new Integer(1));
list.add(new Integer(3));
//list.add("5"); //compiler won't allow this
```

Restrictions on Parametrized Types

- Rules for Java syntax forbid the creation (but not declaration) of an *array* of parametrized Lists:

```
//compiler error
```

```
List<String>[] arrayOfLists = new ArrayList<String>[10];
```

```
/Workaround: can use an ArrayList instead of an Array:
```

```
ArrayList<List<String>> listOfLists =  
    new ArrayList<List<String>>(10);
```

- Subtypes of parametrized types may seem unexpected:

```
ArrayList<Manager> is a subtype of List<Manager>
```

```
ArrayList<Employee> is a subtype of List<Employee>
```

BUT

```
ArrayList<Manager> is NOT a subtype of List<Employee> or  
even of ArrayList<Employee>
```

Generic Subtypes not "Covariant"

1. *Generic Subtyping Is Not Covariant.*

Manager is a subclass of Employee

BUT

ArrayList<Manager> is NOT a subclass of ArrayList<Employee>

2. *Array Subtyping Is Covariant*

Manager is a subclass of Employee

AND

Manager [] is a subclass of Employee []

Miscellaneous Facts About Java Lists

- **Inferred Types in JSE 7 and After:**

When creating an instance of a parametrized type, the parameter can be dropped in the construction step:

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

is the same as:

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

- **Using Lists with Primitives**

- Lists in Java are designed to aggregate *objects*, not primitives.
- Autoboxing allows you to use lists with primitives transparently

```
List<Integer> list = new ArrayList<>();  
list.add(5); //5 converted to Integer type
```

- **Using the keyword var (JSE 10)**

The compiler is able to *infer* the type when you create an instance of a class.

Examples:

```
var list = new ArrayList<Integer>();  
var listOfLists = new ArrayList<List<Integer>>();  
listOfLists.add(list);  
var e = new Employee("Bob", 200000);
```

Exercise 8.4

The code below compiles and produces the output shown. Rewrite the code so that appropriate jse5.0 type parameters are used (like `List<String>` instead of `List`).

```
public class Main {  
    public static void main(String[] args) {  
        List list1 = Arrays.asList("A", "B", "C");  
        List list2 = Arrays.asList("W", "X", "Y");  
        List[] listOfLists = {list1, list2};  
        System.out.println(Arrays.toString(listOfLists));  
    }  
}  
//////Output  
//[["A", "B", "C"], ["W", "X", "Y"]]
```

Outline of Topics

- List ADT, Array Lists and including sort and search
- Linked Lists – singly linked, headers, doubly linked, circular linked
- Generic types for lists before and after jse5.0
- **The List interface, the AbstractList class, and Iterator**
- Collections.sort,
Collections.binarySearch, and
RandomAccess
- **Four ways of Iterating Through Elements in a List**
- Comparators

Java's List Interface

- Both `ArrayList` and `LinkedList` implement the `List` interface in the `Collections` library.
- The operations declared on the `List` interface are identical to the operations in `ArrayList` and `LinkedList` – here is a partial catalogue:

```
interface List<E> {  
    void add(E ob);  
    E get(int pos);  
    boolean remove(E ob);  
    int size();  
    . . .  
}
```

Programming to the Interface

Always type your lists as `List` (as implementers of the `List` interface)

- Supports polymorphism
- Adds flexibility to your implementation

Example: Start with `ArrayList`:

```
List<String> myList = new ArrayList<>();  
myList.add("Bob");  
myList.add("Dave");
```

Later, decide to switch to `LinkedList`:

```
List<String> myList = new LinkedList<>(); //one small change  
myList.add("Bob");  
myList.add("Dave");
```

Using Your List with the Collections API

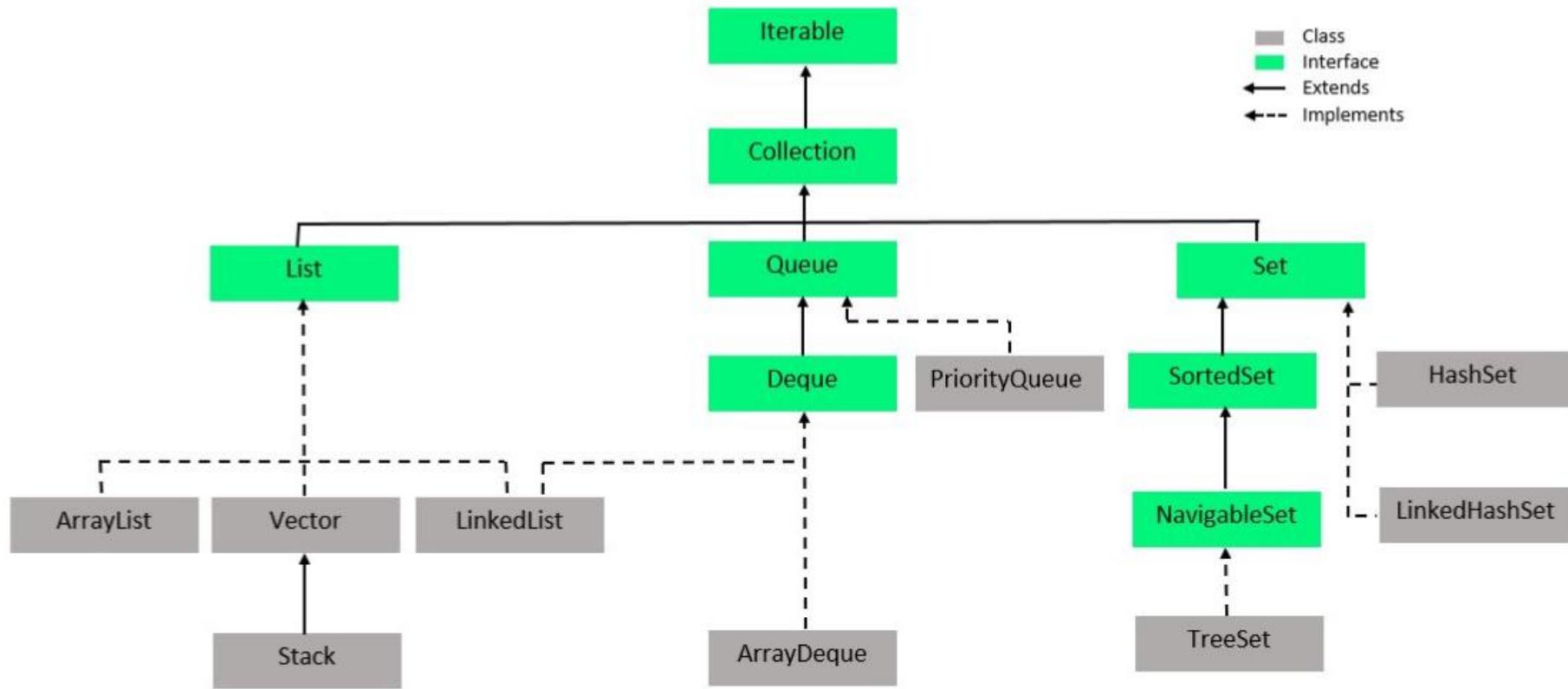
- There is a `Collections` class in the Java library that has many methods like the ones in `Arrays`.

```
Collections.sort(List list)
```

```
Collections.binarySearch(List list)
```

- If you are defining your own list (like `MyStringList`), it is desirable to be able to make use of these methods in `Collections`.
- To use `Collections.sort` or `Collections.binarySearch` with your list, your list must implement the `List` interface

Collection Framework



(continued)

- Every implementer of the `List` interface must implement the super-interface `Iterable`, which means that implementers must provide their own iterators.

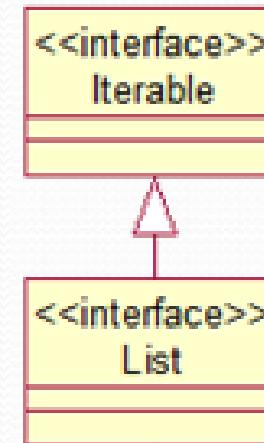
Java's Iterable Interface:

```
interface Iterable {  
    Iterator iterator();  
}
```

Java's Iterator Interface:

`E` is generic type parameter.

```
interface Iterator<E> {  
    boolean hasNext();  
    E next();  
}
```



Java's Iterator

- Since Java's List classes implement the List interface, every type of Java List is equipped with an implementation of Iterator

```
List<String> javaList = new ArrayList<>();  
javaList.add("Bob");  
javaList.add("Steve");  
Iterator<String> it = javaList.iterator();  
while(it.hasNext()) {  
    System.out.println(it.next());  
}
```

See lesson8.iterator

- Implementing an Iterator and using in two ways:
 - Direct use (as above)
 - The for each construct

Sample Implementation of Iterable

Demo: lesson8.demo.MyStringList

```
class MyStringList implements Iterable {
    //. . .
    public Iterator iterator() {
        return new MyIterator();
    }
    private class MyIterator implements Iterator {
        private int position;
        MyIterator() {
            position = 0;
        }
        public boolean hasNext() {
            return (position < size);
        }
        public Object next() throws IndexOutOfBoundsException {
            if(!hasNext()) throw new IndexOutOfBoundsException();
            return strArray[position++];
        }
        public void reset() {
            position = 0;
        }
    }
}
```

(continued)

```
public static void main(String[] args) {  
    var l = new MyStringList();  
    l.add("Bob");  
    l.add("Steve");  
    l.add("Susan");  
    l.add("Mark");  
    l.add("Dave");  
    Iterator iterator = l.iterator();  
    //can explicitly use the iterator  
    while(iterator.hasNext()) {  
        System.out.println(iterator.next());  
    }  
}
```

Using AbstractList with Your List Class

- In addition to the `iterator` method, implementers of `List` must also implement 14 other abstract `List` methods.
- Instead of implementing all the methods in the `List` interface, you can use default implementations provided by the `AbstractList` class.
- `AbstractList` has
 - Two abstract methods: `get(int i)`, `size()`
 - Three methods that need to be overridden: `add`, `remove`, `set` (by default each of these throws an `UnsupportedOperationException`).
- *Big advantage.* Using `AbstractList` as a superclass for your list implementations provides you with an implementation of `Iterator`, saving you from the effort of implementing your own.

Example: Extending AbstractList

(See `lesson8.demo.mystringlist.MyStringListInherit`)

```
//declare your list to extend AbstractList
public class MyStringList extends AbstractList { ... }

public class Test {
    public static void main(String[] args) {
        var l = new MyStringList();
        l.add("Bob");
        l.add("Steve");
        l.add("Susan");
        l.add("Mark");
        l.add("Dave");
        //uses the implementation provided in AbstractList
        var iterator = l.iterator();
        while(iterator.hasNext()) {
            System.out.println(iterator.next());
        }
    }
}
```

Outline of Topics

- List ADT, Array Lists and including sort and search
- Linked Lists – singly linked, headers, doubly linked, circular linked
- Generic types for lists before and after jse5.0
- The List interface, the AbstractList class, and Iterator
- `Collections.sort`,
`Collections.binarySearch`, and
`RandomAccess`
- Four ways of Iterating Through Elements in a List
- Comparators

Using Collections Methods

- Java provides `sort` and `binarySearch` methods for all of its lists (and other types of collections), by way of the `Collections` class.

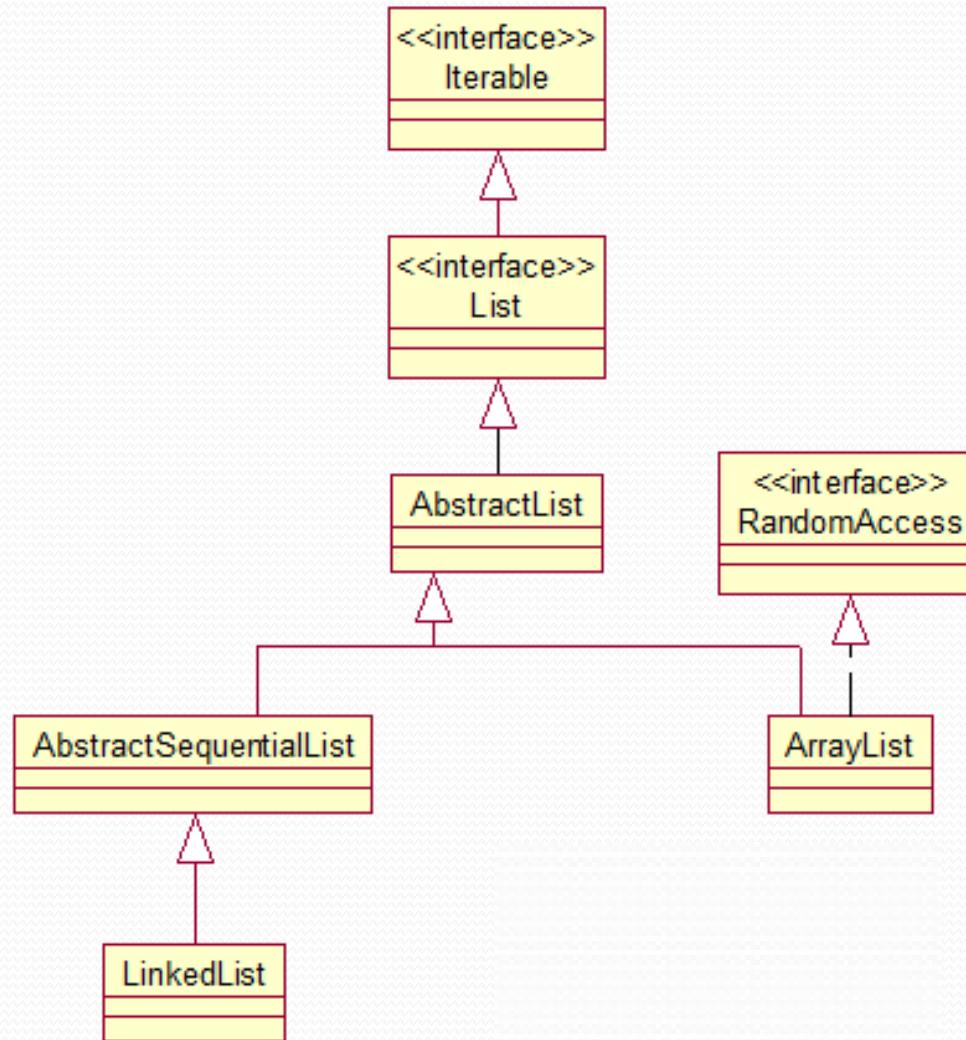
```
List<String> myList = new ArrayList<String>();  
//populate it with a long list of first names, and then...  
Collections.sort(myList);  
int pos = Collections.binarySearch(myList, "Dave");
```

- As you will see in the labs for this lesson, sorting and searching are accomplished in different ways for different lists. It is possible to rewrite `MinSort` in the context of linked lists so that it is approximately as efficient as the `MinSort` for array lists. However, this is not true for binary search. Any known binary search implementation on linked lists is no more efficient than just doing a "find" operation. [This fact is part of the motivation for the invention of Binary Search Trees, which we will discuss later.]

- The reason is that linked lists lack *random access*, so finding the value in the middle of the list is a costly operation. For this reason, Java's `ArrayList` implements a “tag interface” `RandomAccess`. Then, when you call the `binarySearch` method on `Collections` for an `ArrayList`, the method recognizes that the list implements `RandomAccess`, and therefore uses a `binarySearch` implementation discussed in the beginning of this lecture. But if you pass in a `LinkedList`, a slower algorithm is used (since no faster algorithm exists).
- If you want to use the `Collections binarySearch` method on your own array-based list, your list must implement the `List` interface, and, to ensure that the `binarySearch` implementation is efficient, it must also implement the `RandomAccess` interface.

See

`LectureCode/lesson8/demo/mystringlist/MyStringListInheritRandom.java`



Summary for ArrayList and LinkedList

ArrayList	Linked list
Fixed size of background array: Resizing is expensive	Dynamic size
Insertions and Deletions are inefficient: Background array must be reconstructed frequently	Insertions and Deletions are efficient since these require only small changes in links
Random access i.e., efficient indexing	No random access Not suitable for operations requiring repeated access of elements by index such as sorting
No memory waste if the array is full or almost full; otherwise may result in much memory waste.	Since memory is allocated dynamically, there is no waste of memory.

Main Point

An Array List encapsulates the random access behavior of arrays, and incorporates automatic resizing and optionally may include support for sorting and searching. Using a style of sequential access instead, Linked Lists improve performance of insertions and deletions, but at the cost of losing fast element access by index.

Random and sequential access provide analogies for forms of gaining knowledge. Knowledge by way of the intellect is always sequential, requiring steps of logic to arrive at an item of knowledge. Knowing by intuition (*prathibha*) , or by way of *ritam-bhara pragya*, is knowing the truth without steps – a kind of “random access” mode of gaining knowledge.

Outline of Topics

- List ADT, Array Lists and including sort and search
- Linked Lists – singly linked, headers, doubly linked, circular linked
- Generic types for lists before and after jse5.0
- The List interface, the AbstractList class, and Iterator
- Collections.sort,
Collections.binarySearch, and
RandomAccess
- **Four ways of Iterating Through Elements in a List**
- Comparators

Iterating Through Elements in a List

- Demo: lesson8.traverse
- First way – Using for loops

```
List<String> list = new ArrayList<>();  
.....  
String next = null;  
for(int i = 0; i < list.size(); ++i) {  
    next = list.get(i);  
    //do something with next  
}
```

Iterating Through Elements in a List

- Second way – Using Iterator

```
String next = null;  
Iterator<String> iterator = list.iterator();  
while(iterator.hasNext()) {  
    next = iterator.next();  
    //do something  
}
```

Iterating Through Elements in a List

- Third way – Using for each construct (the list has to implement Iterable in order to use for each construct)

```
//use the for each construct, which uses an  
//iterator in the background
```

```
for(String str : list) {  
    //do something  
}
```

Iterating Through Elements in a List

- Fourth way – Using Java 8's New `forEach` Function
- A default method `forEach` was added to the `Iterable` interface. Consequently, any Java library class that implements `Iterable`, as well as any user-defined class that implements `Iterable`, has automatic access to this new method.
- The `forEach` method takes a lambda expression of the form
`x -> function(x)` where `function(x)` does not return a value.

- Examples:

```
//Java's List  
List<String> javaList  
    = new ArrayList<>();  
javaList.add("Bob");  
javaList.add("Carol");  
javaList.add("Steve");  
  
javaList.forEach(  
    name -> System.out.println(name)  
);  
  
//output  
Bob  
Carol  
Steve
```

```
//User-defined list that  
//implements Iterable  
MyStringList list  
    = new MyStringList();  
list.add("Bob");  
list.add("Carol");  
list.add("Steve");  
  
list.forEach(  
    name -> System.out.println(name)  
);  
  
//output  
Bob  
Carol  
Steve
```

Exercise 8.5 -- forEach

- Use the new `forEach` method of `Iterable` to produce the list of names of all `Employees` contained in a given input list.

```
public class Main {  
    public static void main(String[] args) {  
        List<Employee> aList = Arrays.asList(new Employee("Bob", 20000),  
            new Employee("Harry", 60000), new Employee("Steven", 30000),  
            new Employee("Regis", 50000),new Employee("Tony", 40000));  
        System.out.println(empsToNames(aList));  
    }  
  
    static List<String> empsToNames(List<Employee> list) {  
        ..  
    }  
}
```

Outline of Topics

- List ADT, Array Lists and including sort and search
- Linked Lists – singly linked, headers, doubly linked, circular linked
- Generic types for lists before and after jse5.0
- The List interface, the AbstractList class, and Iterator
- Collections.sort,
Collections.binarySearch, and
RandomAccess
- Four ways of Iterating Through Elements in a List
- Comparators

Comparing Objects for Sorting and Searching

- Java supports sorting of many types of objects. To sort a list of objects, it is necessary to have some “ordering” on the objects. For example, there is a natural ordering on numbers and on Strings. But what about a list of Employee objects?
- In practice, we may want to sort business objects in different ways. An Employee list could be sorted by name, salary, or hire date.

Employee Data		
Name	Hire Date	Salary
Joe Smith	11/23/2000	50000
Susan Randolph	2/14/2002	60000
Ronald Richards	1/1/2005	70000

- Implementing the Comparable interface allows you to sort a list of Employees with reference to one primary field – for instance, you could sort by name or by salary, but you do not have the option to change this primary field.
- A more flexible interface for such requirements is provided by the **Comparator** interface, whose only method is **compare()**. Like lists, in j2se5.0, Comparators are parametrized.

```
public interface Comparator<T> {  
    int compare(T o1, T o2);  
}
```

- The compare method is expected to behave in the following way:

For objects a and b,

- `compare(a, b)` returns a negative number if `a` is “less than” `b`
- `compare(a, b)` returns a positive number if `a` is “greater than” `b`
- `compare(a, b)` returns 0 if `a` “equals” `b`

Example

```
public class Employee {  
    private String name;  
    private int salary;  
  
    public Employee(String name,int aSalary) {  
        this.name = name;  
        salary = aSalary;  
    }  
    public String getName() {  
        return name;  
    }  
    public int getSalary() {  
        return salary;  
    }  
    public String toString() {  
        return name + " : " + salary;  
    }  
}
```

```
public class NameComparator implements Comparator<Employee>{  
    @Override  
    public int compare(Employee e1, Employee e2) {  
        return e1.getName().compareTo(e2.getName());  
    }  
}  
public class SalaryComparator implements Comparator<Employee>{  
    @Override  
    public int compare(Employee e1, Employee e2) {  
        if(e1.getSalary()<e2.getSalary()) return -1;  
        else if(e1.getSalary()>e2.getSalary()) return 1;  
        else return 0;  
    }  
}
```

```
public class Main {  
    public static void main(String[] args) {  
        Employee[] arr = {new Employee("Bob", 2000),  
                          new Employee("Steve", 3000),  
                          new Employee("Alice", 50000)};  
        System.out.println(Arrays.toString(arr));  
        Arrays.sort(arr, new NameComparator());  
        System.out.println(Arrays.toString(arr));  
        Arrays.sort(arr, new SalaryComparator());  
        System.out.println(Arrays.toString(arr));  
    }  
} ///////////////output:  
// [Bob: 2000, Steve: 3000, Alice: 50000]  
// [Alice: 50000, Bob: 2000, Steve: 3000]  
// [Bob: 2000, Steve: 3000, Alice: 50000]
```

Exercise 8.6 – Sorting a Collection of Lists

```
public class Main {  
  
    public static void main(String[] args) {  
        List<String> list1 = dataList1();  
        List<String> list2 = dataList2();  
        List<String> list3 = dataList3();  
  
        //Step 1: Sort each list  
  
        //Step 2: Assemble the sorted lists into a single collection of lists  
  
        //Step 3: Sort the combined list using a Comparator. Declare that  
        //listA comes before listB if the 0th element of A precedes the 0th  
        //element of B. Then print the combined list to the console  
  
    }  
}
```

Comparators and the compare Contract

The compare contract for Comparators:

It *must* be true that:

- a is “less than” b if and only if b is “greater than” a
- if a is “less than” b and b is “less than” c, then a must be “less than” c.

It *should* also be true that the Comparator is *consistent with equals*; in other words:

- `compare(a, b) == 0 if and only if a.equals(b)`

If a Comparator is not consistent with equals, problems can arise when using different container classes. For instance, the `contains` method of a Java List uses `equals` to decide if an object is in a list. However, containers that maintain the order relationship among elements (like TreeSet – more on this one later) check whether the output of `compare` is 0 to implement `contains`. See demos `lesson8.consisequals` and `lesson8.consisequals_contains`

Example: A Name Comparator

Demo: lesson8.comparator

```
// Assumes Employee contains just name and hireDate as
// instance variables
public class NameComparator implements Comparator<Employee> {
    //is this implementation consistent with equals?
    public int compare(Employee e1, Employee e2) {
        return e1.getName().compareTo(e2.getName());
    }
}
public class EmployeeSort {
    public static void main(String[] args) {
        new EmployeeSort();
    }
    public EmployeeSort() {
        Employee[] empArray =
            {new Employee("George", 1996, 11, 5),
             new Employee("Dave", 2000, 1, 3),
             new Employee("Richard", 2001, 2, 7)};
        List<Employee> empList = Arrays.asList(empArray);
        Comparator<Employee> nameComp = new NameComparator();
        Collections.sort(empList, nameComp);
        System.out.println(empList);
    }
}
public class Employee {
    private String firstName;
    private Date hireDate;
    // ...
}
```

Question

- How can the Comparator in the previous example be made consistent with equals?

Solution

```
public class NameComparator implements  
    Comparator<Employee> {  
    // consistent with equals  
    public int compare(Employee e1, Employee e2) {  
        String name1 = e1.getName();  
        String name2 = e2.getName();  
        Date hireDate1 = e1.getHireDay();  
        Date hireDate2 = e2.getHireDay();  
        if(name1.compareTo(name2) != 0) {  
            return name1.compareTo(name2);  
        }  
        //in this case, name1.equals(name2) is true  
        return hireDate1.compareTo(hireDate2);  
    }  
}
```

Connecting the Parts of Knowledge With the Wholeness of Knowledge

All knowledge contained in point

1. An implementation of the abstract class AbstractSequentialList in Java (such as a LinkedList) results in a list that has only sequential access to its elements.
2. An implementation of the RandomAccess interface in Java (such as ArrayList and Vector) results in a list that has random access (and therefore, effectively, instantaneous access) to its elements.

3. **Transcendental Consciousness:** TC is the home of all knowledge. All knowledge has its basis in the unbounded field of pure consciousness.
4. **Wholeness moving within itself:** In Unity Consciousness, when the home of all knowledge has become fully integrated in all phases of life, it is possible to know anything, any particular thing, instantly.