
MAHARISHI UNIVERSITY of MANAGEMENT
Engaging the Managing Intelligence of Nature
Computer Science Department
CS390 Fundamental Programming Practices (FPP)
Professor Anne McCollum

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`

The List Abstract Data Type

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

- Typical operations:

find(Object o)	returns position of first occurrence
findKth(int pos)	returns element based on index
insert(Object o, int pos)	inserts object into specified position
remove(Object o)	removes object
printList()	outputs all elements
makeEmpty()	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`

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. (Recall prog7-2 where recursive minsort was implemented.)
- `MinSort` can use 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.

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 using the binary search strategy described 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

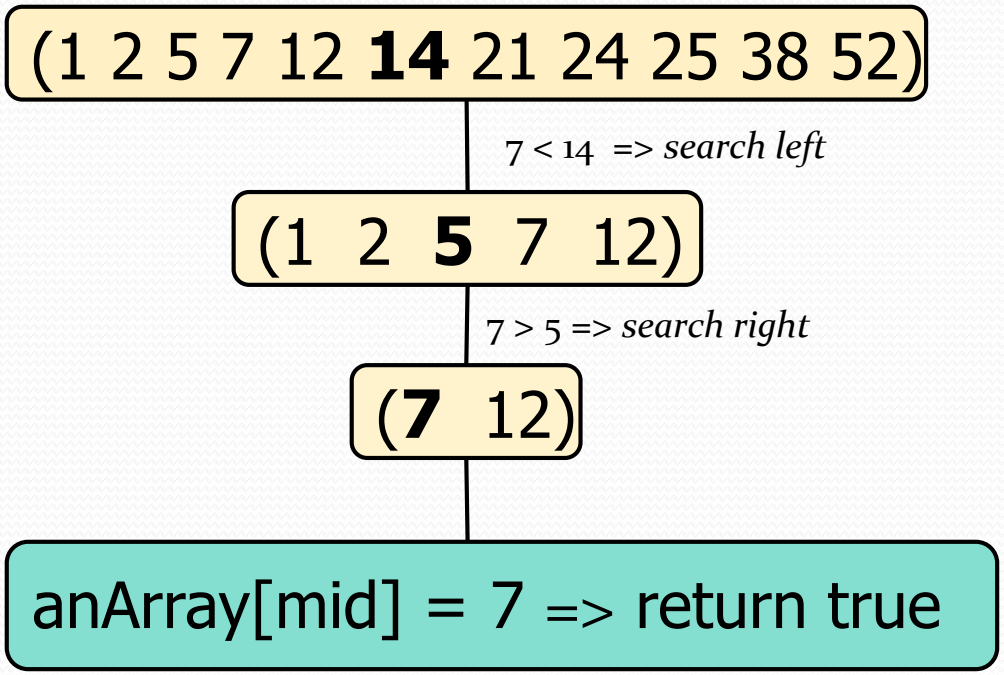
Online pictorial demo: http://www.cs.armstrong.edu/liang/animation/web/BinarySearch.html

Here is the code for binary search, applied to sorted arrays.

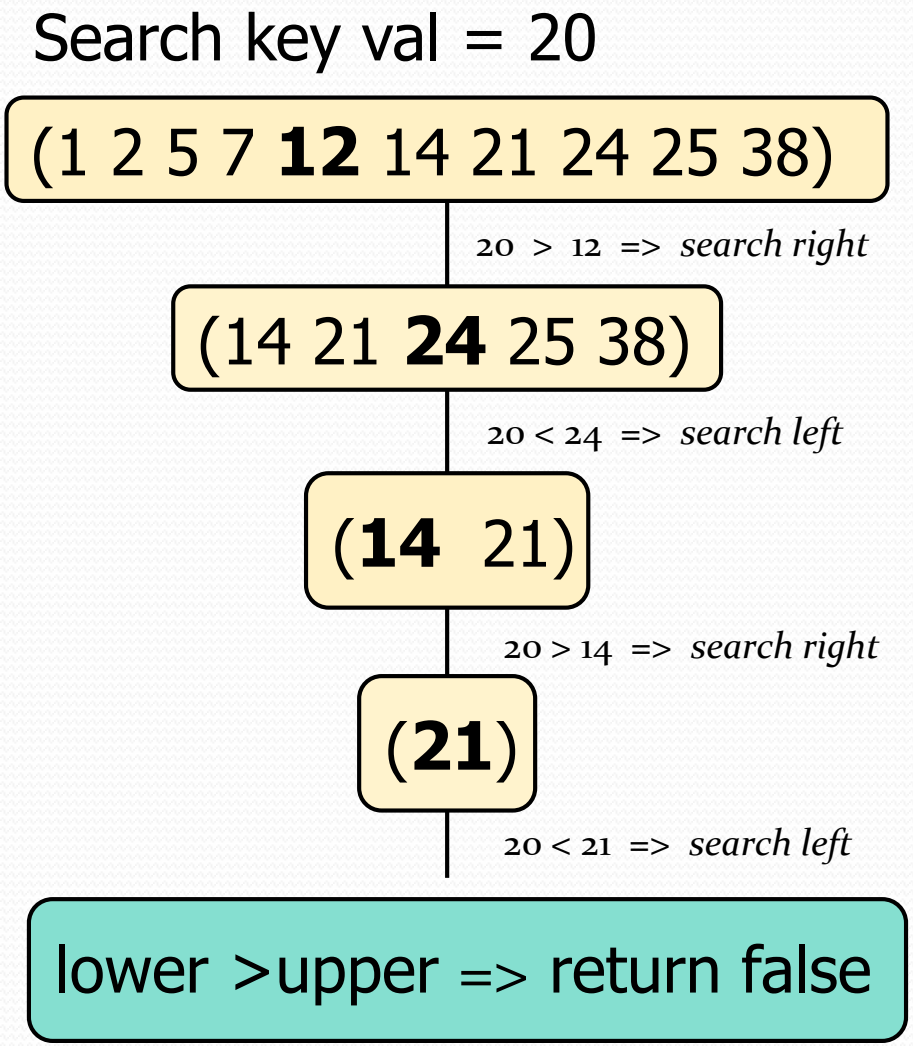
```
int[] anArray;  //instance variable
public boolean search(int val) {
   boolean b = recSearch(0, anArray.length-1, val);
   return b;
}

private boolean recSearch(
        int lower, int upper, int val) {
   int mid = (lower + upper)/2;
   if (anArray[mid] == val) return true;
   if (lower > upper) return false;
   if (val > anArray[mid]) {
      return recSearch(mid + 1, upper, val);
   }
   return recurse(lower, mid - 1, val);
}
```

Example

Search key val = 7



Example


Search key val = 20
(1 2 5 7 **12** 14 21 24 25 38)
20 > 12 => *search right*
(14 21 **24** 25 38)
20 < 24 => *search left*
(**14** 21)
20 > 14 => *search right*
(**21**)
20 < 21 => *search left*
lower >upper => return false

Searching a Sorted Array 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`.

ArrayList `insert`, `add`, and `remove` Operations are Inefficient

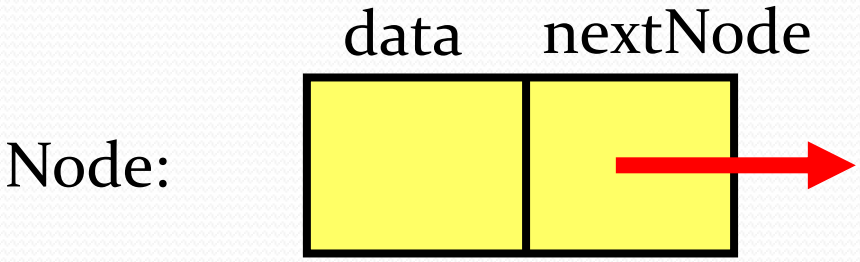
If, in using an `ArrayList`, the operations `remove`, `insert`, and `add` are used predominantly, performance is not optimal because of repeated resizing and other steps that require array copying. For such purposes, another implementation of List is better.

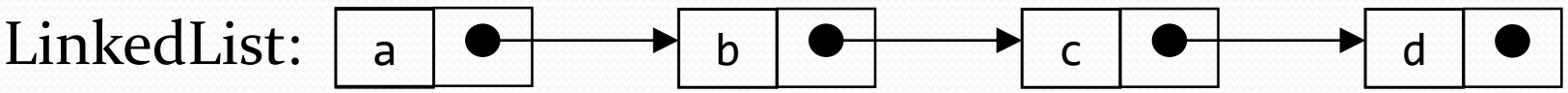
Outline of Topics

- 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

```
// definition of Node
// for most of the code
public class Node {
    String data;
    Node nextNode;
}
```

Adding Nodes to the LinkedList

If the linked list had only one element it would be quite simple. It would have a first node which is created as the start node, that is filled with data.

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

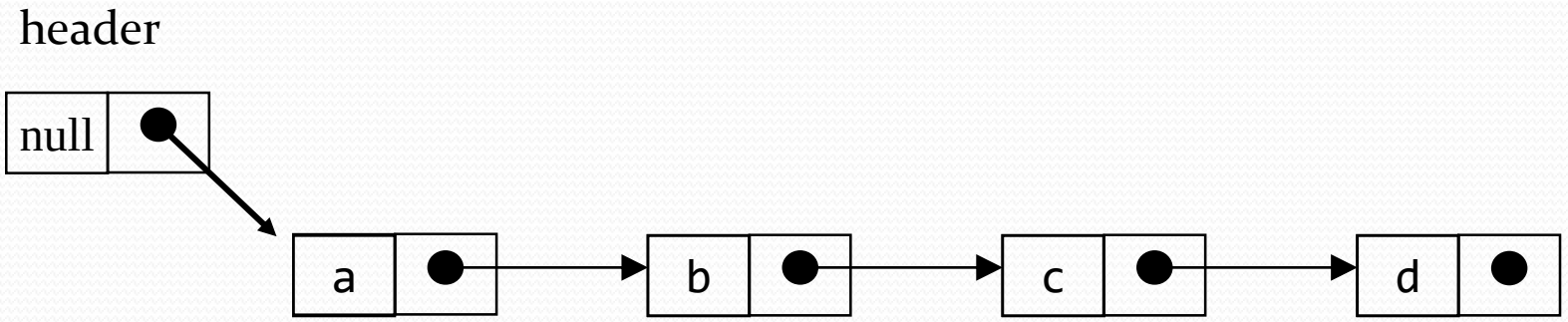
System.out.println(startNode) for the above list would print out: A

When the list becomes more than one node, a link needs to be created between the new node and the node(s) already in the list. In the example below, the link between nodes is set in the statement "startNode.node = next":

```
// Two Node List
Node newNode = new Node();
newNode.data = "B";
startNode.nextNode = newNode;
```

System.out.println(startNode) for the above list would print out: A B

Linked Lists with Headers



- 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. The header gives you a starting point for searching, modifying, or printing the list.

- Demo: `lesson8.singlylinked`

Populating LinkedList Nodes using addFirst()

LectureCode demo lesson8.singlylinked shows a LinkedList with a no data header. Populating the list by adding elements from the end of the array using addFirst avoids the need to go to the end of the list each time an element is added.

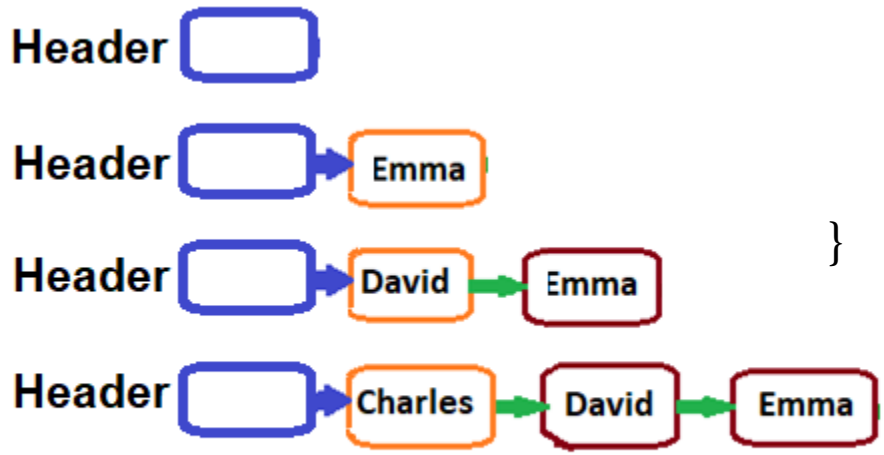
```
public class SinglyLinkedListWithHeader {
    Node header = null; //contains no data, never removed

     public SinglyLinkedListWithHeader() {
        //header should never be null
        header = new Node();
     }
     String[] stringData = {"Albert", "Billy", "Charles", "David", "Emma"};
       SinglyLinkedListWithHeader sll = new
     SinglyLinkedListWithHeader();
       //populate - start adding from the END of the array
       for(int i = stringData.length -1; i >= 0; i--) {
         sll.addFirst(stringData[i]);
       }
```

The start of a picture showing this list:

Populating Nodes in a LinkedList using addFirst()

lesson8.singlylinked has a no data header. Populating the linked list starts by adding elements from the end of the startData array (on previous slide) to the front of the linked list. (This avoids the need to traverse to the end of the list each time you add an element.)

The previous slide initialized the header with this code:
Node header = new Node();



Continuing in this way...

```
void addFirst(String s) {
    Node newNode = new Node();
    newNode.data = s;

    //link newNode to current header.nextNode
    newNode.nextNode = header.nextNode;

    //link from header to newNode
    header.nextNode = 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. (In some implementations, a link to the node at the end is maintained.)

```
// definition of Node
// for this code
public class Node {
    String data;
    Node next;
}
```

```
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;

    }
}
```

Insert Nodes into a LinkedList

lesson8.singlylinked continued. Inserting into the linked list requires first finding the right position in the linked list. (This is covered later.)

```
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); //same as add(s);
    else if(pos == 0) addFirst(s);
    else {
        Node n = new Node();
        n.data = s;
        if(header.node == null) header.node = n;
        else {
            Node temp = header.node;
            int count = 1;
            while(count < pos) {
                temp = temp.node;
                count++;
            }
            //insert n between temp and temp.node
            Node last = temp.node;
            temp.node = n;
            n.node = last;  //placing n into position pos
        }
    }
}
```

`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 *addLast* both require traversing the `Node`s 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

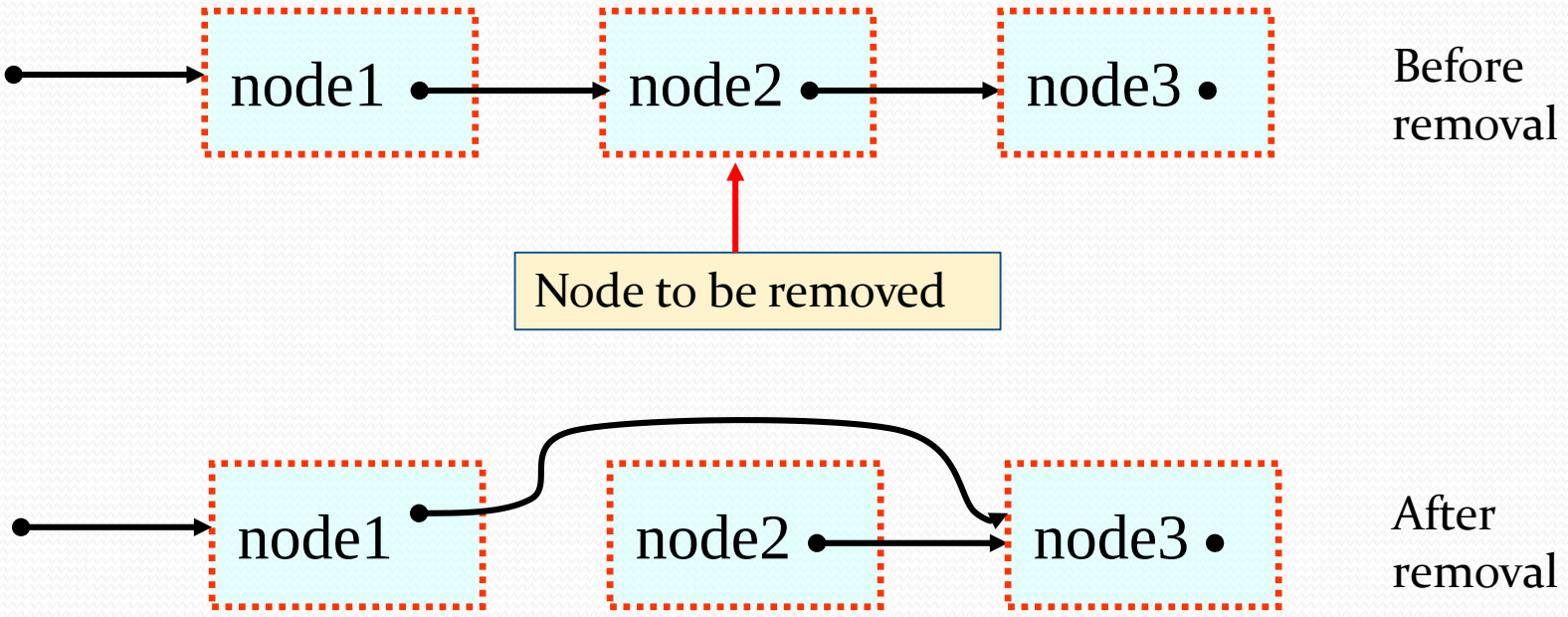
Implementing *search* Operation

- *search* requires traversing the nodes via links, starting at the first node, either till value is found or till next node is null

```
// definition of Node
public class Node {
   String data;
   Node nextNode;
}
```

```
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.nextNode;
   }
   return false;
}
```

Implementing *remove* Operation



Implementing *remove* Operation

An important step in `remove` is being able to finding the previous node.

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

```
// definition of Node
// for this code
public class Node {
   String data;
   Node nextNode;
}
```

```
void remove(String s) {
    if(s == null) return;
    Node next = header.nextNode;
    Node previous = header;
    //if next == null, s cannot be removed
    while(next != null) {
    if(s.equals(next.data)) {
        previous.nextNode = next.nextNode;
        return;
    }
    previous = next;
    next = next.nextNode;
    }
}
```

Implementing the *size* operation.

Involves traversing nodes following links and keeping count of how many nodes have been visited.

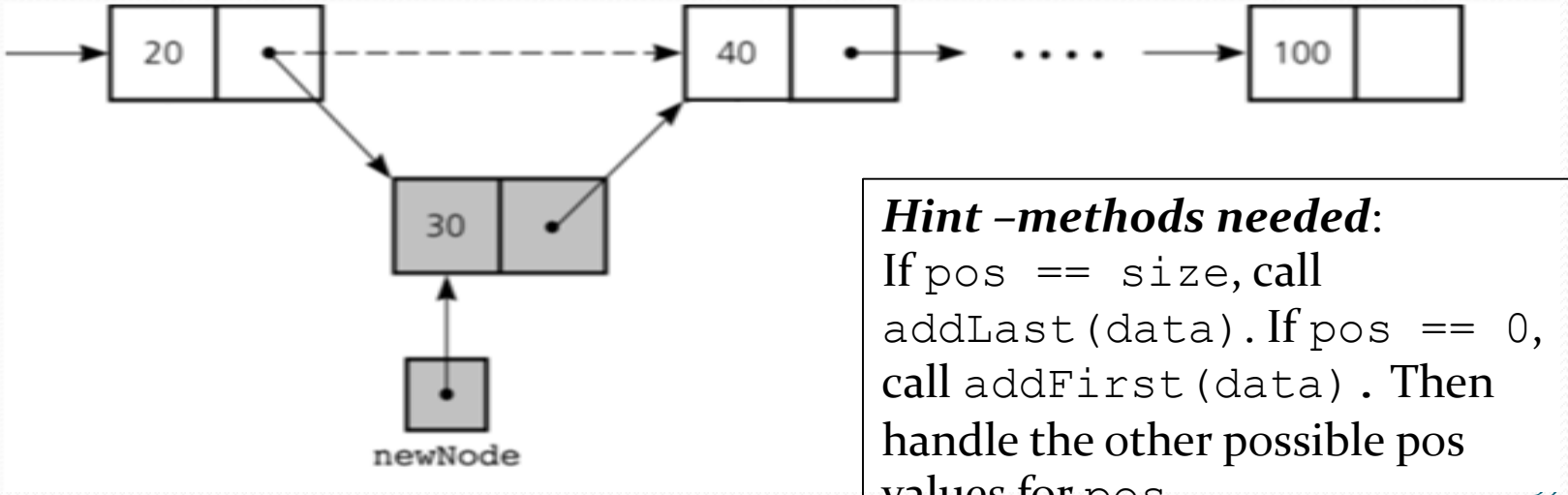
```
// definition of Node
public class Node {
   String data;
   Node nextNode;
}
```

```
/** 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.nextNode;
  }
  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`

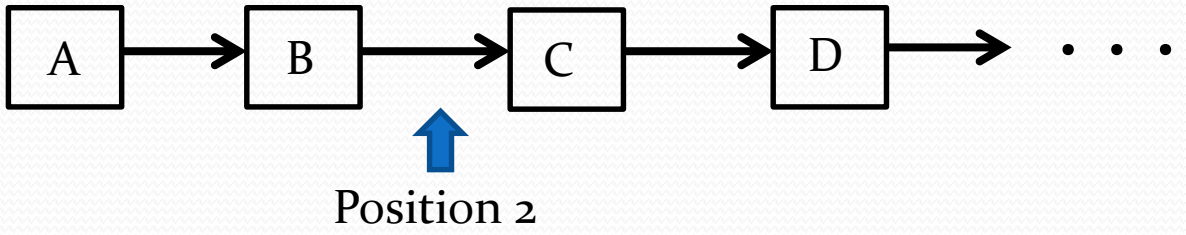
```
void insert(String data, int pos)
```



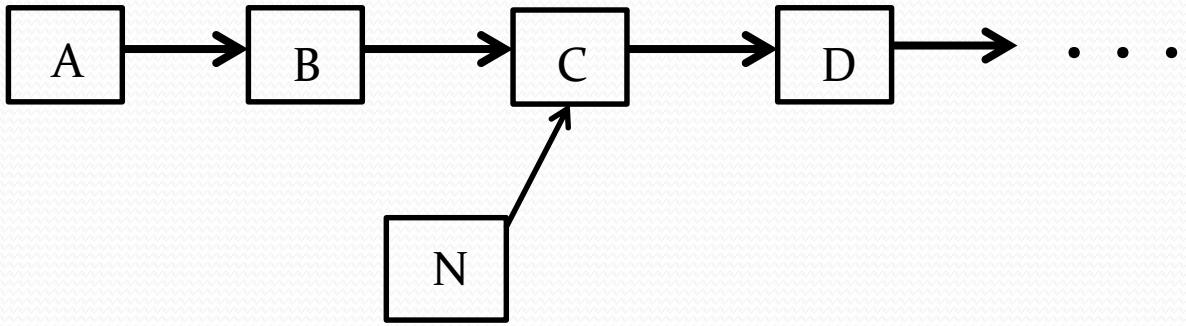
Hint –methods needed:
If `pos == size`, call `addLast(data)`. If `pos == 0`, call `addFirst(data)`. Then handle the other possible pos values for `pos`.

Exercise 8.3: Steps to Implement `insert` Operation

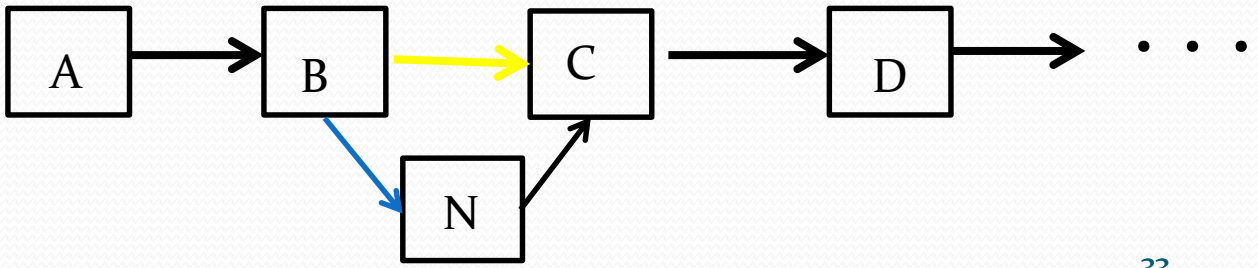
1. Locate position to insert



New node steps:
2. Create new node N
3. Set N's link next



Existing node steps:
5. Reassign existing node B next link to N



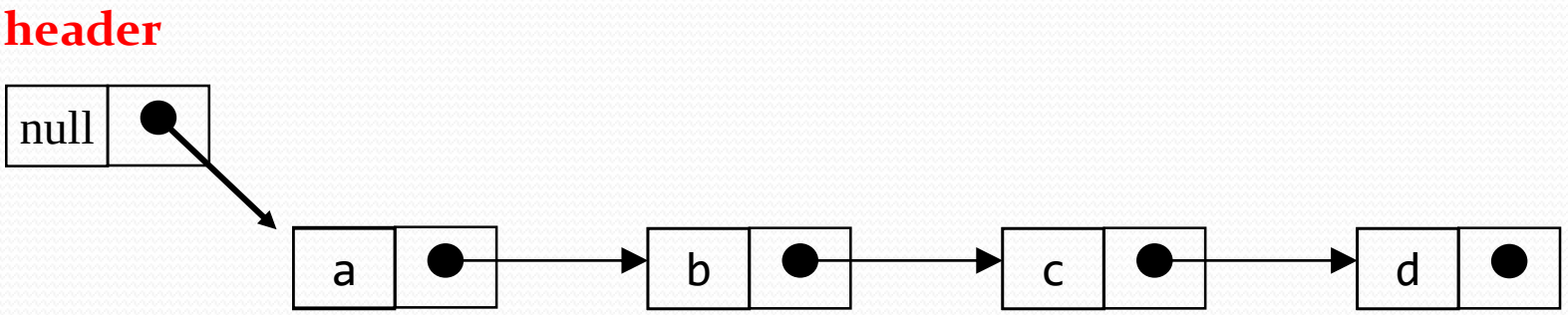
Solution

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

```
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); //same as add(s);
   else if (pos == 0) addFirst(s);
   else {
      // Create new node N
      Node n = new Node();
      n.data = s;
      //startNode will not be null at this point
      Node previous = startNode;
      // Locate position to insert
      for(int i = 0; i < pos - 1; ++i) {
           previous = previous.nextNode;
      }
      //insert between previous and previous.nextNode
      //placing n into position pos
      n.nextNode = previous.nextNode;
      previous.nextNode = n;
   }
}
```

Note: These methods use the LinkedList instance variables and methods

Linked Lists with Headers

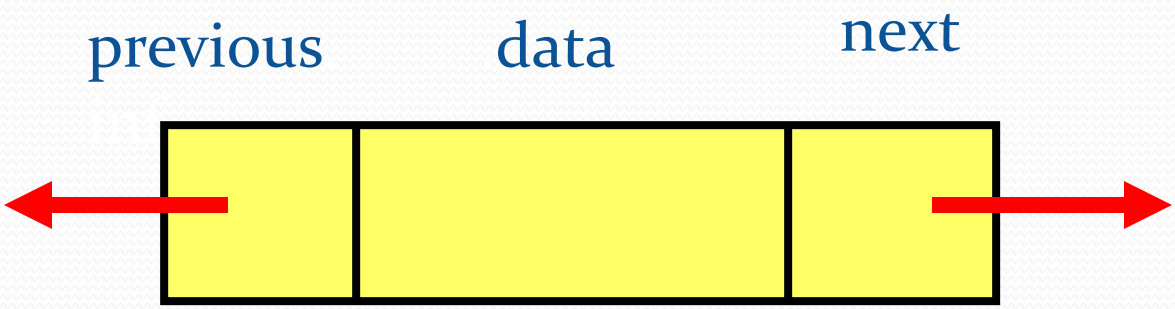


Reminder: A singly linked list is usually implemented using a header. A header is a Node that contains no data, can never be removed, and has a link to first Node.

(`lesson8.singlylinked`)

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.



Implementation of Doubly Linked List

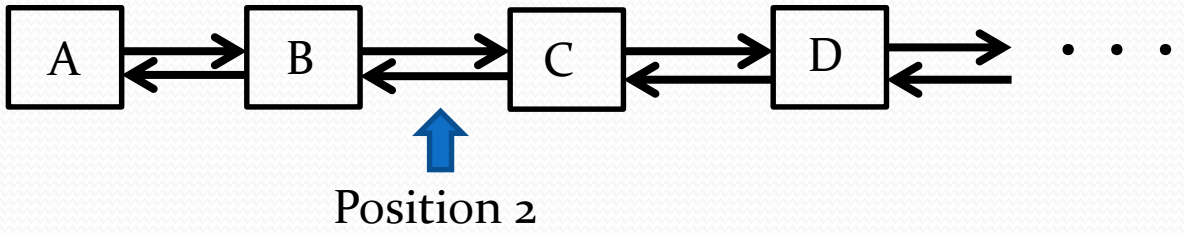
```
public class MyStringLinkedList {
    Node header;
    MyStringLinkedList(){
        header = new Node(null,null, null);
    }
    public void addFirst(String item){
        Node n = new Node(header.next,header,item);
        if(header.next != null){
            header.next.previous = n;
       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

    }
    class Node {
        String value;
        Node next;
        Node previous;
        Node(Node next, Node previous, String value){
            this.next = next;
            this.previous = previous;
            this.value = value;
        }
     }
```

Question: How to implement the `insert` and `remove` operation?

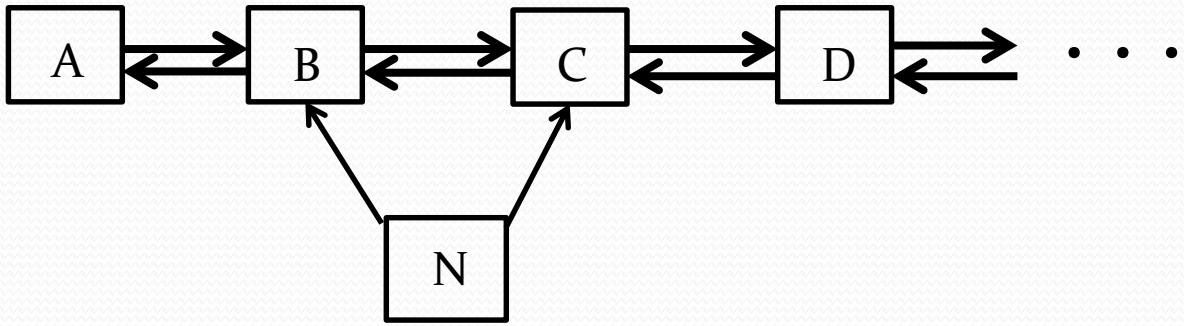
- See Demo (`lesson8.doublylinked.DoublyLinkedList`)

Inserting at a Specified Position

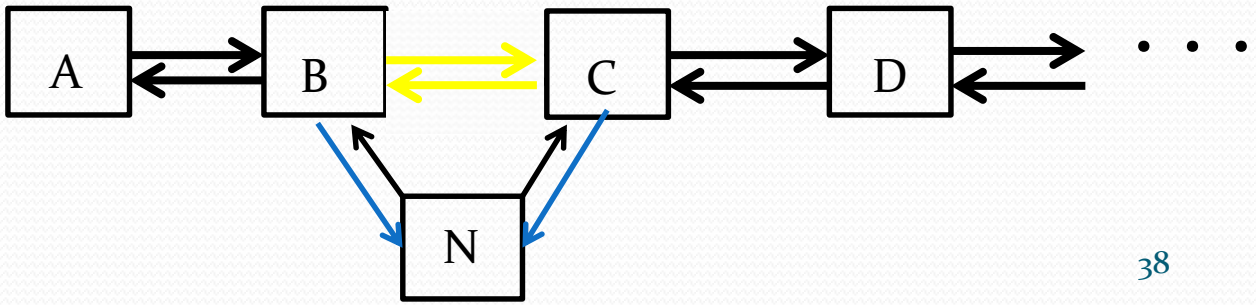
1. Locate position to insert



New node steps:
2. Create new node N
3. Set N's link next
4. Set N's link previous

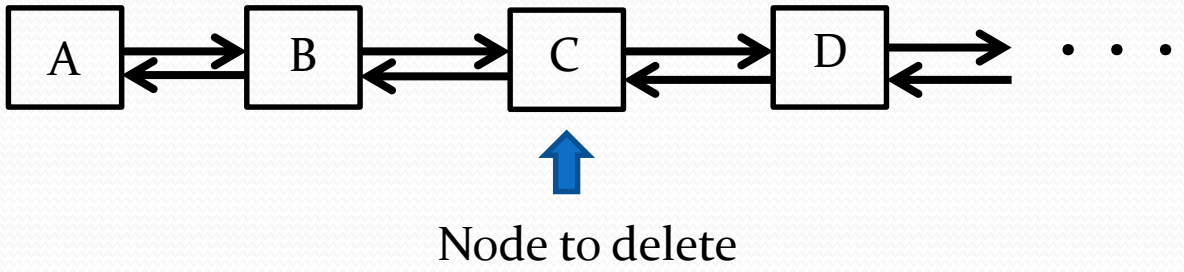


Existing node steps:
5. Reassign existing node B next link to N
6. Reassign existing node C previous link to N



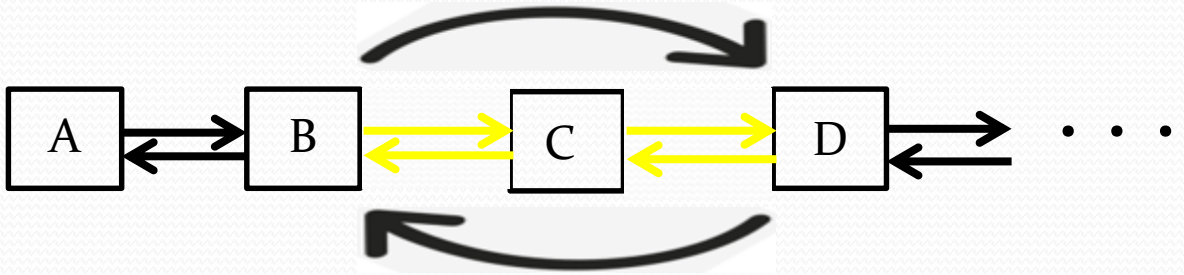
Deleting a Node

1. Locate position of node C to be deleted.



Break links into and out of node C by:

2. setting the B next link (which is the same as C.previous link) to D Node
3. setting the D.previous link (which is the same as C.next link) to D Node



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.

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 (legacy code) 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`

Objects Stored in a List

`MyStringList` and `MyStringLinkedList` work only if the objects we wish to store are `Strings`. It is very time consuming and inefficient to rewrite the List code for each type as the need arises (`MyEmployeeList`, `MyIntegerList`, `MyAccountList`).

Better to create one List that stores any element. In old code this was done by having the List store elements of type `Object`.

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-based list that accepts any type of object.

  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.

  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);
```

A runtime exception occurs here because there is no compiler checking of types in a collection. (Why is this pattern undesirable?)

- From j2se5.0 on, Lists include a generic parameter. Here are declarations from the Java library (where `E` stands for a generic type parameter):

```
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> listStr = new ArrayList<String>();
listStr.add("Bob");
listStr.add("Sally");
String name = listStr.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
```

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 automates the adding of primitives into Lists

```
List<Integer> list = new ArrayList<>();
list.add(5);  //5 is automatically 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 legacy 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]]
```

Exercise 8.4 - Solution

Only <String> needs to be added to the List declaration.

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

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 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.

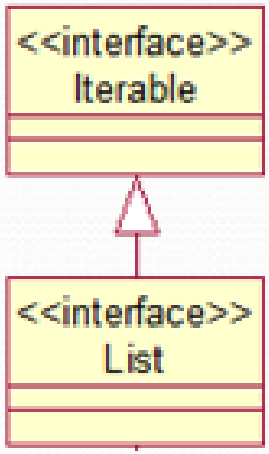
Note: The java.util.List interface in the JDK gives full details on everything that must be implemented, including the information that follows.

(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 (implicit)

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
  - One abstract method: `get(int i)`
  - 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

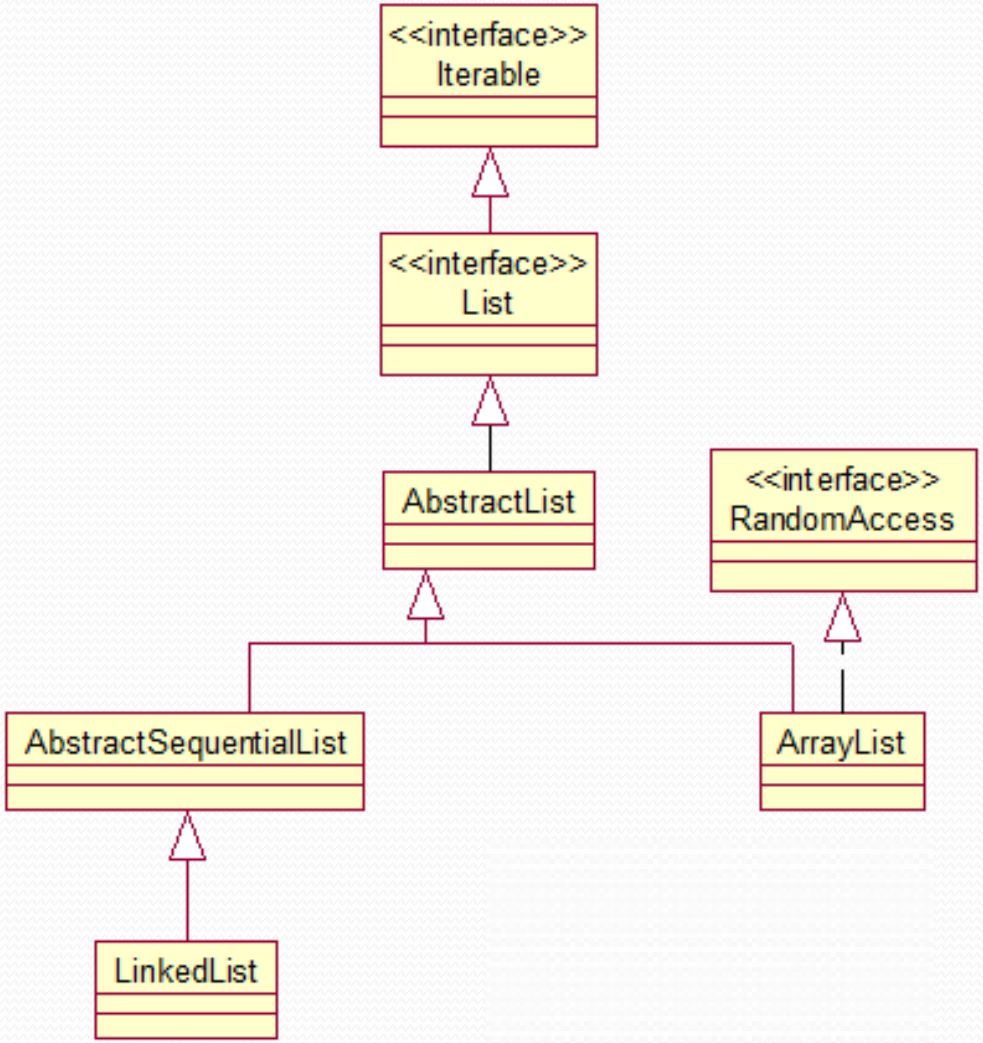
- Java provides `sort` and `binarySearch` methods for all of its lists (and other types of collections), by way of the `Collections` class.

```
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.]

- 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. If you pass in a `LinkedList`, a slower algorithm is usually used. (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


<<interface>>
Iterable
<<interface>>
List
AbstractList
<<interface>>
RandomAccess
AbstractSequentialList
ArrayList
LinkedList

Using Collections Methods

Notice that in the previous slide, the ArrayList is an implementation of the List interface. The ArrayList can use the Collections sorting methods, because it implements the List interface. The same is true for any list that wishes to use the Collections sorting methods.

Modifier and Type	Method	Description
static <T extends Comparable<? super T>> void	sort(**List**<**T**> list)	Sorts the specified list into ascending order, according to the natural ordering of its elements.
static <T> void	sort(**List**<**T**> list, Comparator<? super T> c)	Sorts the specified list according to the order induced by the specified comparator.

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 accessing 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.

The random and sequential access modes give 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 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 an enhanced or advanced for loop construct. (This used to be called a "for each" loop, before lambda expressions.) Note: The list has to implement `Iterable` in order to use the advanced for construct.

```
//use the advanced for loop 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, numbers and `Strings` have a natural ordering. 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 `Employee`s with reference to one 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(), which takes two**. Like lists, in j2se5.0, `Comparators` are parametrized.

```
public interface Comparator<T> {
   int compare(T o1, T o2);
}
(T represents an object, any object.)
```

- 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

```java
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;
    }
}
```

```java
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]
```

```java
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;
    }
}
```

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
    }
}
```

Exercise 8.6 – Solution to Sort Collection of Lists

```
Public class Main {
  static void main(String[] args) {
     List<String> list1 = dataList1();
     List<String> list2 = dataList2();
     List<String> list3 = dataList3();

     //Step 1: Sort each list
     Collections.sort(list1);
     Collections.sort(list2);
     Collections.sort(list3);

     //Step 2: Assemble the sorted lists into a single collection of lists
     List<List<String>> all = Arrays.asList(list1, list2, list3);

     //Step 3: Sort combined list using a Comparator. Print combined list to console
     Collections.sort(all, new ListComparator());
     System.out.println(all);
  }
}

static class ListComparator implements Comparator<List<String>> {
     @Override
     public int compare(List<String> l1, List<String> l2) {
          String s1 = l1.get(0);
          String s2 = l2.get(0);
          return s1.compareTo(s2);
     }
}
```

ListComparator determines the sort order

`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, they should return the same result for the same input data:

- **`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 demo `lesson8.consisequals`

Example: A Name Comparator

Demo: `lesson8.comparator`

```
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;
    // . . .
   public boolean equals(Object ob) {
      if(ob == null) return false;
      if(!(ob instanceof Employee)) return false;
      Employee e = (Employee)ob;
      return e.name.equals(name) && e.hireDay.equals(hireDay);
   }
}
```

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) 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.