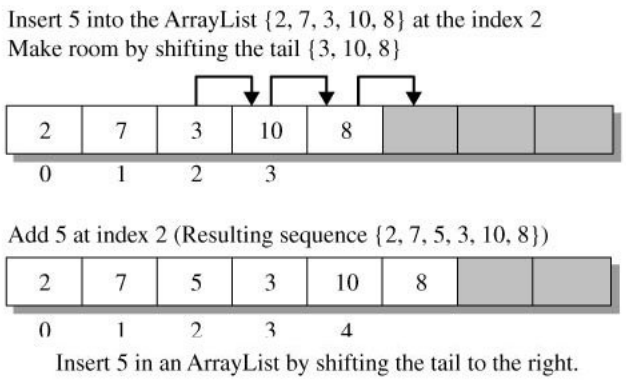
Linked Lists

Introducing Linked Lists

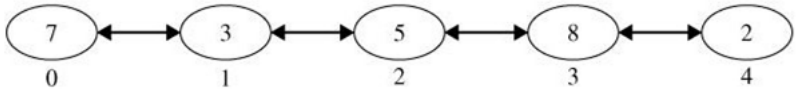
- To insert or remove an element at an interior location in an ArrayList requires shifting of data and is an O(n) operation.



Insert 5 in an ArrayList by shifting the tail to the right.

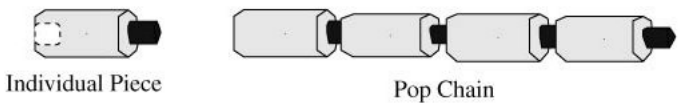
Introducing Linked Lists (continued)

- We need an alternative structure that stores elements in a sequence but allows for more efficient insertion and deletion of elements at random positions in the list. In a linked list , elements contain links that reference the previous and the successor elements in the list.



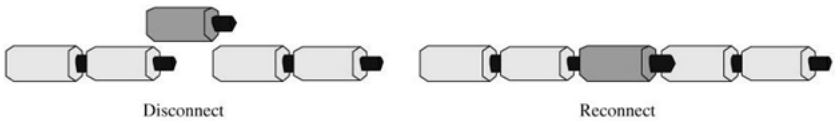
Introducing Linked Lists (continued)

- Think of each element in a linked list as being an individual piece in a child's pop chain. To form a chain, insert the connector into the back of the next piece



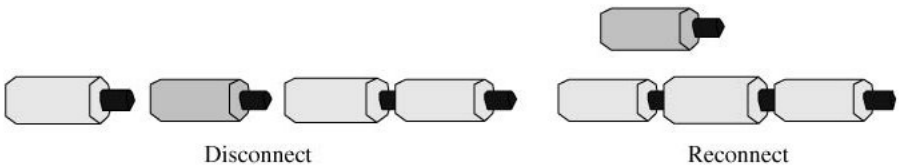
Introducing Linked Lists (continued)

- Inserting a new piece into the chain involves merely breaking a connection and reconnecting the chain at both ends of the new piece.



Introducing Linked Lists (continued)

- Removal of a piece from anywhere in the chain requires breaking its two connections, removing the piece, and then reconnecting the chain.



Introducing Linked Lists (concluded)

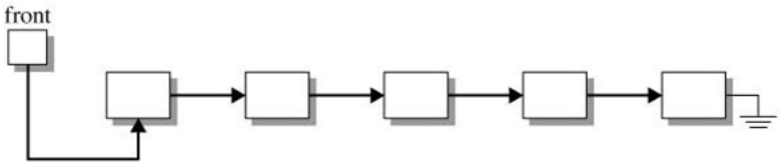
- Inserting and deleting an element is a local operation and requires updating only the links adjacent to the element. The other elements in the list are not affected. An ArrayList must shift all elements on the tail whenever a new element enters or exits the list.

Structure of a Linked List

- Each element is a *node* that consists of a value and a reference (link) to the next node in the sequence.
- A node with its two fields can reside anywhere in memory.
- The list maintains a reference variable, front, that identifies the first element in the sequence. The list ends when the link null ( ).

Structure of a Linked List (concluded)

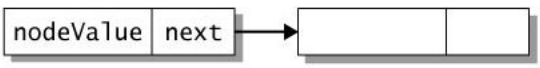
- A singly-linked list is not a direct access structure. It must be accessed sequentially by moving forward one node at a time



Singly linked list where an element is a node.

Creating a Linked List

- Elements in a linked list are nodes. These are Node objects that have two instance variables. The first variable, **`nodeValue`**, is of generic type T. The second variable is a Node reference called **`next`** that provides a link to the next node



Creating a Linked List (continued)

- Linked lists are implementation structures and so Node objects are rarely visible in the public interface of a data structure. As a result, we declare the instance variables in the Node class public. This greatly simplifies the writing of code involving linked lists.
- The Node class is a *self-referencing* structure, in which the instance variable, **`next`**, refers to an object of its own type.

Creating a Linked List The Node Class

- The class has two constructors that combine with the new operator to create a node. The default constructor initializes each instance variable to be **`null`**. The constructor with an type parameter initializes the **`nodeValue`** field and sets next to null.

Creating a Linked List
The Node Class

```
public class Node<T>
{
   // data held by the node
   public T nodeValue;
   // next node in the list
   public Node<T> next;

   // default constructor with no initial value
   public Node()
   {
      nodeValue = null;
      next = null;
   }
```


Creating a Linked List
The Node Class

```
   // initialize nodeValue to item and set next to null
   public Node(T item)
   {
      nodeValue = item;
      next = null;
   }
}
```

Creating a Linked List (continued)

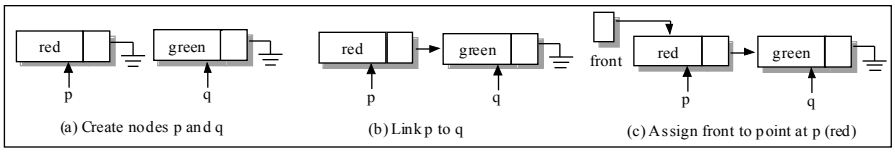
- Need a reference variable, front, that identifies the first node in the list.
- Once you are at the first node, you can use next to proceed to the second node, then the third node, and so forth.

Creating a Linked List (continued)

- Create a two element linked list where the nodes have string values "red" and "green". The variable front references the node "red". The process begins by declaring three Node reference variables front, p, and q.

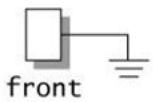
Creating a Linked List (continued)

```
Node<String> front, p, q;      // references to nodes
p = new Node<String>("red");   // create two nodes (figure (a)
q = new Node<String>("green");

// create the link from p to q by assigning the next field
// for node p the value q
p.next = q;                    // figure (b)

// set front to point at the first node in the list
front = p;                     // figure (c)
```



Creating a Linked List (concluded)

- If a linked list is empty, front has value null.



Node<T> front = null

Scanning a Linked List

- Scan a singly linked list by assigning a variable curr the value of front and using the next field of each node to proceed down the list. Conclude with curr == null.
- As an example of scanning a list, the static method toString() in the class ds.util.Nodes takes front as a parameter and returns a string containing a comma-separated list of node values enclosed in brackets.

Nodes.toString()

```
public static <T> String toString(Node<T> front)
{
   if (front == null)
      return "null";

   Node<T> curr = front;
   // start with the left bracket and
   // value of first node
   String str = "[" + curr.nodeValue;
```

Nodes.toString() (concluded)

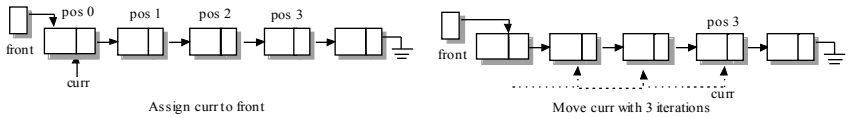
```
   // append all but last node, separating
   // items with a comma polymorphism calls
   // toString() for the nodeValue type
   while(curr.next != null)
   {
      curr = curr.next;
      str += ", " + curr.nodeValue;
   }
   str += "]";
   return str;
}
```


Locating a List Position

- To locate an element at position n, we need to scan the list through a specified number of node.
- Declare a Node reference curr to point at the first element (front) of the list. This is position 0. A for-loop moves curr down the sequence n times. The variable curr then references the element at position n. The value at position n is curr.nodeValue.

Locating a List Position (concluded)

```
Node<T> curr = front;
// move curr down the sequence through n successor nodes
for (int i = 0; i < n; i++)
       curr = curr.next;
```



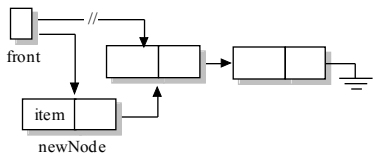
Updating the Front of the List

- Inserting or deleting an element at the front of a list is easy because the sequence maintains a reference that points at the first element.

Updating the Front of the List (continued)

- To insert, start by creating a new node with **`item`** as its value. Set the new node to point at the current first element. Then update front to point at the new first element.

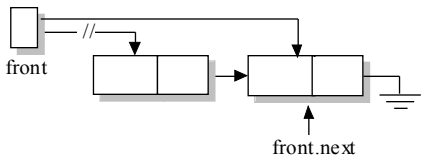
```
Node<T> newNode = new Node<T>(item);

// insert item at the front of the list
newNode.next  = front;
front = newNode;
```



Updating the Front of the List (concluded)

- Deleting the first element involves setting front to reference the second node of the list.

```
front = front.next;      // establish a new front
```

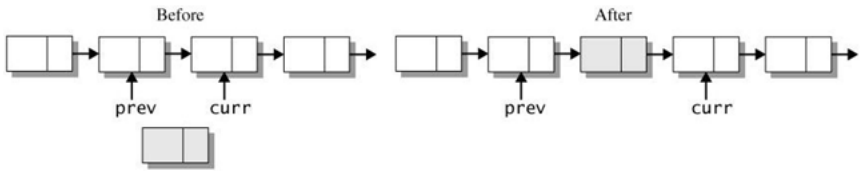


General Insert Operation

- Inserting a new node before a node referenced by curr involves updating only adjacent links and does not affect the other elements in the sequence.

General Insert Operation (continued)

- To insert the new node before a node referenced by curr, the algorithm must have access to the predecessor node prev since an update occurs with the next field of prev.

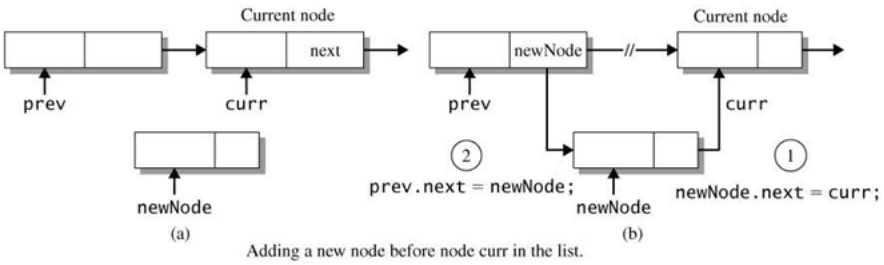


General Insert Operation (continued)

- Create newNode with value item.
- Connecting newNode to the list requires updating the values of newNode.next and prev.next.

General Insert Operation (concluded)

```
Node curr, prev, newNode;
// create the node and assign it a value
newNode = new Node(item);
// update links
newNode.next = curr;       // step 1
prev.next = newNode;       // step 2
```



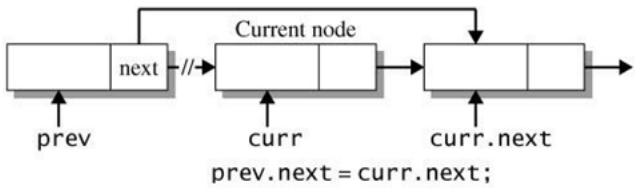
Adding a new node before node curr in the list.

General Delete Operation

- Deleting the node at position curr also requires access to the predecessor node prev.
- Update the link in the predecessor node by assigning prev to reference the successor of curr (curr.next).

General Delete Operation (concluded)

```
Node curr, prev;

// reconnect prev to curr.next
prev.next = curr.next;
```



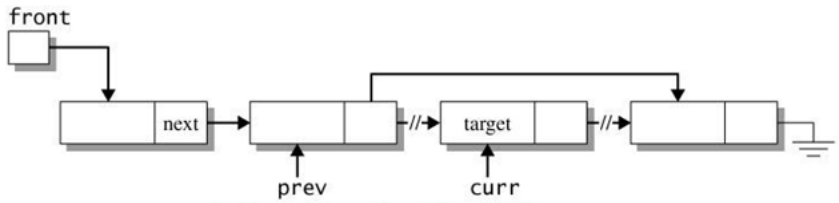
Removing a node at position curr in the list.

Removing a Target Node

- To remove the first occurrence of a node having a specified value, begin with a scan of the list to identify the location of the target node.
- The scan must use a pair of references that move in tandem down the list. One reference identifies the current node in the scan, the other the previous (predecessor) node.

Removing a Target Node (continued)

- Once curr identifies the node that matches the target, the algorithm uses the reference prev to unlink curr.



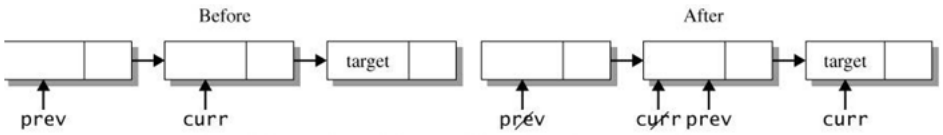
Positions of prev and curr for a deletion.

Removing a Target Node (continued)

- Set reference curr to the front of the list and prev to null, since the first node in a linked list does not have a predecessor.
- Move curr and prev in tandem until curr.nodeValue matches the target or curr == null.

```
prev = curr;        // update prev to next position (curr)
curr = curr.next;   // move curr to the next node
```


Removing a Target Node (continued)



Update of references prev and curr in the scan of nodes in a linked list.

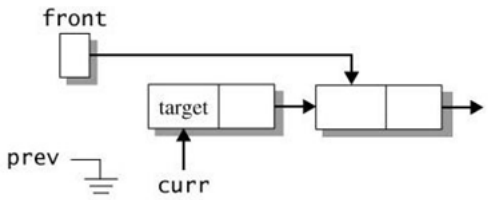
Removing a Target Node (continued)

- If the scan of the list identifies a match (target.equals(curr.nodeValue)), curr points at the node that we must remove and prev identifies the predecessor node.
- There are two possible situations that require different actions. The target node might be the first node in the list, or it might be at some intermediate position in the list. The value of prev distinguishes the two cases.

Removing a Target Node (continued)

- *Case 1*: Reference prev is null which implies that curr is front. The action is to delete the front of the list.

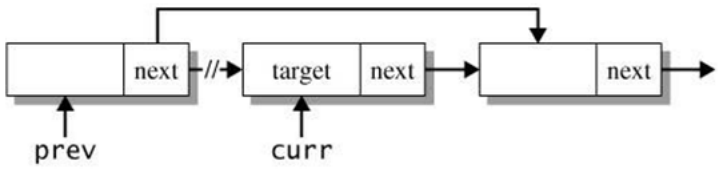
```
front = curr.next;
```



Removing a Target Node (continued)

- *Case 2*: The match occurs at some intermediate node in the list. Both curr and prev have non-null values. The action is to delete the current node by unlinking it from prev.

```
prev.next = curr.next;
```



Removing a Target Node (concluded)

- The method generic remove() has a parameter list that includes a reference to the front of the list and the target value.
- The method returns the value of front, which may have been updated if the deletion occurs at the first node in the list.

remove() Method

```
// delete the first occurrence of the target in the
// linked list referenced by front; returns the
// value of front
public static <T> Node<T> remove(Node<T> front,
T target)
{
   // curr moves through list, trailed by prev
   Node<T> curr = front, prev = null;
   // becomes true if we locate target
   boolean foundItem = false;
```


remove() Method (continued)

```
   // scan until locate item or come to end of list
   while (curr != null && !foundItem)
   {
      // check for a match; if found, check
      // whether deletion occurs at the front
      // or at an intermediate position
      // in the list; set boolean foundItem true
      if (target.equals(curr.nodeValue))
      {
         // remove the first Node
         if (prev == null)
            front = front.next;
         else
            // erase intermediate Node
            prev.next = curr.next;
         foundItem = true;
      }
```

remove() Method (concluded)

```
      else
      {
         // advance curr and prev
         prev = curr;
         curr = curr.next;
      }
   }
   // return current value of front which is
   // updated when the deletion occurs at the
   // first element in the list
   return front;
}
```


Program 10.1

```
import java.util.Random;
import java.util.Scanner;
import ds.util.Node;
// methods toString() and remove()
import ds.util.Nodes;

public class Program10_1
{
   public static void main(String[] args)
   {
      // declare references; by setting front to null,
      // the initial list is empty
      Node<Integer> front = null, newNode, p;

      // variables to create list and
      // setup keyboard input
      Random rnd = new Random();
      Scanner keyIn = new Scanner(System.in);
      int listCount, i;
```

Program 10.1 (continued)

```
// prompt for the size of the list
System.out.print("Enter the size of the list: ");
listCount = keyIn.nextInt();

// create a list with nodes having random
// integer values from 0 to 99; insert
// each element at front of the list
for (i = 0; i < listCount; i++)
{
   newNode = new Node<Integer>(rnd.nextInt(100));
   newNode.next = front;
   front = newNode;
}

System.out.print("Original list:  ");
System.out.println(Nodes.toString(front));
```


Program 10.1 (continued)

```
System.out.print("Ordered list:   ");
// continue finding the maximum node and
// erasing it until the list is empty
while (front != null)
{
   p = getMaxNode(front);
   System.out.print(p.nodeValue + "  ");
   front = Nodes.remove(front, p.nodeValue);
}
System.out.println();
}
```

Program 10.1 (continued)

```
   // return a reference to the node
   // with the maximum value
   public static <T extends Comparable<? super T>>
   Node<T> getMaxNode(Node<T> front)
   {
      // maxNode reference to node
      // containing largest value (maxValue);
      // initially maxNode is front and
      // maxValue is front.nodeValue; scan
      // using reference curr starting with
      // the second node (front.next)
      Node<T> maxNode = front, curr = front.next;
      T maxValue = front.nodeValue;
```


Program 10.1 (concluded)

```
      while (curr != null)
      {
         // see if maxValue < curr.nodeValue;
         // if so, update maxNode and maxValue;
         // continue scan at next node
         if (maxValue.compareTo(curr.nodeValue)< 0)
         {
            maxValue = curr.nodeValue;
            maxNode = curr;
         }
         curr = curr.next;
      }
      return maxNode;
   }
}
```

Program 10.1 (Run)

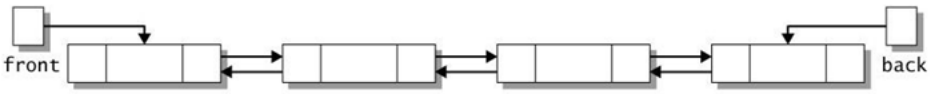
```
Run:

Enter the size of the list: 9
Original list:  [77, 83, 14, 38, 70, 35, 55, 11, 6]
Ordered list:   83  77  70  55  38  35  14  11  6
```


Doubly Linked Lists

- Doubly-linked list nodes contain two references that point to the next and previous node.
- Such a list has a reference, **`front`**, that points to the first node in the sequence and a reference, **`back`**, that points at the last node in the sequence.

Doubly Linked Lists (continued)

- You can scan a doubly-linked list in both directions. The forward scan starts at front and ends when the link is a reference to back. In the backward direction simply reverse the process and the references.



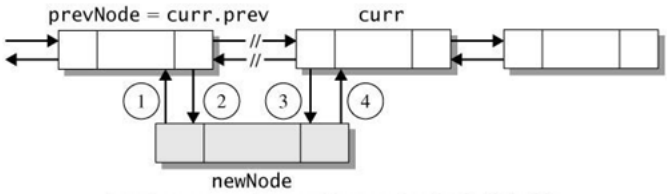
Doubly Linked Lists (continued)

- Like a singly-linked list, a doubly-linked list is a sequential structure.
- To move forward or backward in a doubly-linked list use the node links next and prev.
- Insert and delete operations need to have only the reference to the node in question.

Doubly Linked Lists (continued)

- Inserting into a doubly linked list requires four reference assignments.

```
prevNode = curr.prev;
newNode.prev = prevNode;   // statement 1
prevNode.next = newNode;   // statement 2
curr.prev = newNode;       // statement 3
newNode.next = curr;       // statement 4
```

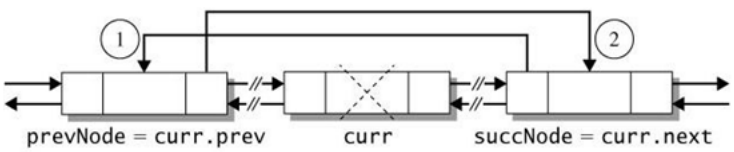


Inserting a new node at position curr in a doubly linked list.

Doubly Linked Lists (continued)

- To delete a node curr, link the predecessor (curr.prev) of curr to the successor of curr (curr.next).

```
prevNode = curr.prev;
succNode = curr.next;
succNode.prev = prevNode;  // statement 1
prevNode.next = succNode;  // statement 2
```



Doubly Linked Lists (concluded)

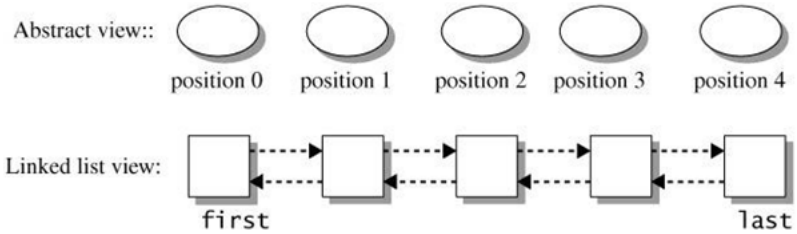
- In a singly-linked list, adding and removing a node at the front of the list are O(1) operation.
- With a doubly-linked list, you can add and remove a node at the back of the list with same runtime efficiency. Simply update the reference back.

The LinkedList Collection

- An ArrayList uses an array in contiguous memory.
- A LinkedList uses a doubly-linked list whose elements reside in noncontiguous memory locations.
- View a LinkedList collection as a disjoint sequence of nodes starting at the first element and proceeding in successive order to a final element. Motion can be forward or backward from a given node.

The LinkedList Collection (continued)

- The figure provides views of a LinkedList collection as an abstract sequence and as a linked list.



The LinkedList Collection (continued)

- The LinkedList class has a default constructor that creates an empty list.
- A toString() method returns a string representing the list as a comma-separated sequence of elements enclosed in brackets.

The LinkedList Collection (continued)

- By implementing the List interface, the class also implements the Collection interface with its familiar general purpose collection methods isEmpty(), size(), contains(), and toArray().
- The collection add() method inserts an element. Since a LinkedList allows duplicates, add() will always insert a new element at the back of the list and return true.

The LinkedList Collection (concluded)

- A call to remove() with an Object reference deletes the first occurrence of the object in the list. The method returns true or false depending on whether a match occurs.

LinkedList Example

1. **Use the constructor to create an empty linked list.**

```
LinkedList<String> aList = new LinkedList<String>();
```

2. **Assume the list contains the strings "Red", "Blue", "Green". Output its size and check whether aList contains the color "White".**

```
System.out.println("Size = " +  aList.size());
System.out.println("List contains the string 'White' is " +
                                  aList.contains("White");
```

```
Output:  Size = 3
         List contains the string 'White' is false
```


LinkedList Example (concluded)

Add the color "Black" and a second element with color "Blue". Then delete the first occurrence of "Blue". An output statement uses toString() to list the elements in the sequence.

```
aList.add("Black");           // add Black at the end
aList.add("Blue");            // add Blue at the end
aList.remove("Blue");         // delete first "Blue"
System.out.println(aList);    // uses toString()
```

```
Output: [Red, Green, Black, Blue]
```

LinkedList Index Methods

- The LinkedList class implements the indexed-based methods that characterize the List interface.
- A collection can access and update an element with the **`get()`** and **`set()`** methods and modify the list with the **`add(index, element)`** and **`remove(index)`** methods.
- The index methods have O(n) worst case running time. Use these methods only for small data sets.

LinkedList Index Methods Example

Assume the collection, list, initially contains elements with values [5, 7, 9, 4, 3].

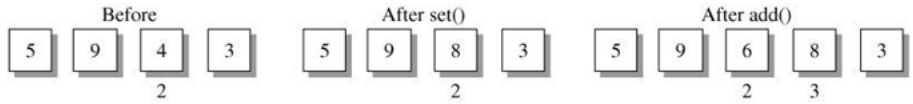
Use get() to access the object at index 1 and then remove the element. The element at index 1 then has the value 9.

```
Integer intObj = list.get(1); // intObj has value 7
list.remove(1);
```

LinkedList Index Methods Example (concluded)

Use set() to update the element at index 2. Give it the value 8. Add a new element with value 6 at index 2. The new element occupies position 2, and its insertion shifts the tail of the list up one position. Thus the node at index 3 has value 8.

```
list.set(2, 8);
list.add(2, 6);
```



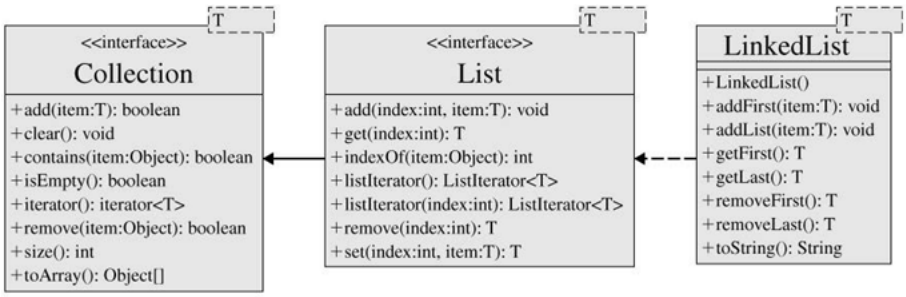
Accessing the Ends of a LinkedList

- A series of O(1) operations access and update the elements at the ends of the list.
- For the front of the list, the class defines the methods getFirst(), addFirst(), and removeFirst().
- The counterparts at the back of the list are getLast(), addLast(), and removeLast().

Accessing the Ends of a LinkedList (concluded)

- A linked list is a natural storage structure for implementing a queue. The element at the front (getFirst()) is the one that exits (removeFirst()) the queue. A new element enters (addLast()) at the back of the queue.

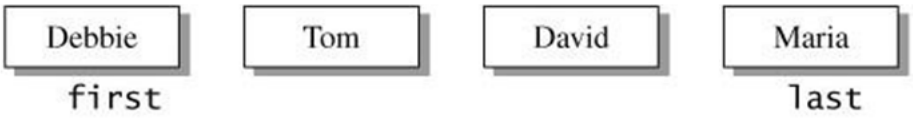
UML for the LinkedList Class



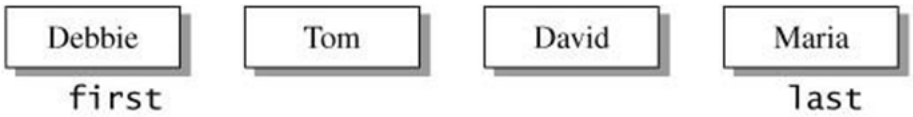
End-of-List Methods Example

The "add" methods build the list by adding a new element. Observe that successive calls to addFirst() inserts elements in reverse order; successive calls to addLast() inserts the elements in the normal order.

```
list.addFirst("Tom");
list.addFirst("Debbie");

list.addLast("David");
ist.addLast("Maria");
```



End-of-List Methods Example (continued)



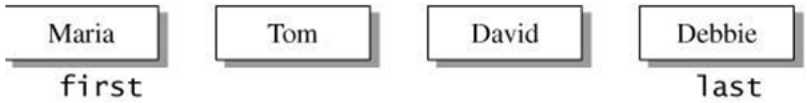

```
// identify the elements at the ends of the list
System.out.println("First element is " + list.getFirst());
System.out.println("Last element is " + list.getLast());
```

```
Output:  First element is Debbie
         Last element is Maria
```

End-of-List Methods Example (continued)

Exchange the first and last elements in the list.

```
String firstElement, lastElement;
// remove the elements at the ends of the list and capture
// their values
firstElement = aList.removeFirst();
lastElement = aList.removeLast();
// add the elements back into the list with firstElement
// at the back and lastElement at the front
aList.addLast(firstElement);
aList.addFirst(lastElement);
```



End-of-List Methods Example (concluded)

Output the elements in the list by position. Repeatedly delete the first element and display its value until the list is empty.

```
while (!aList.isEmpty())
   System.out.print(aList.removeFirst() + "  ");
```

```
Output:
   Maria  Tom  David  Debbie
```

Program 10.2

```
import ds.util.LinkedList;
import java.util.Scanner;

public class Program10_2
{
   public static void main(String[] args)
   {
      // create an empty linked list
      LinkedList<String> draftlist =
         new LinkedList<String>();

      // variables used to update the draft list
      int fromIndex, toIndex;
      char updateAction;
      String playerName;
      String obj;
```


Program 10.2 (continued)

```
      // initial names in the list and the
      // keyboard input file
      String[] playerArr ={"Jones", "Hardy",
         "Donovan", "Bundy"};
      Scanner keyIn = new Scanner(System.in);
      String inputStr;

      // initialize the list
      for (int i = 0; i < playerArr.length; i++)
         draftlist.add(playerArr[i]);

      // give instructions on updating the list
      System.out.println("Add player:    " +
         "Input 'a' <name>");
      System.out.println("Shift player:  " +
         "Input 's' <from> <to>");
      System.out.println("Delete player: " +
         "Input 'r' <name>" + "\n");
```

Program 10.2 (continued)

```
// initial list
System.out.println("List: " + draftlist);

// loop executes the simulation of draft updates
while (true)
{
   // input updateAction, exiting on 'q'
   System.out.print("   Update: ");
   updateAction = keyIn.next().charAt(0);

   if (updateAction == 'q')
      break;

   // execute the update
   switch(updateAction)
   {
```


Program 10.2 (continued)

```
      case 'a':
         // input the name and add to end of list
         playerName = keyIn.next();
         draftlist.add(playerName);
         break;

      case 'r':
         // input the name and remove from list
         playerName = keyIn.next();
         draftlist.remove(playerName);
         break;

      case 's':
         // input two indices to shift an
         // element from a source position
         // to a destination position;
         // remove element at source and
         // add at destination
         fromIndex = keyIn.nextInt();
```

Program 10.2 (concluded)

```
                // set to list position
                fromIndex--;
                toIndex = keyIn.nextInt();
                // set to list position
                toIndex--;
                obj = draftlist.remove(fromIndex);
                draftlist.add(toIndex, obj);
                break;
            }
            // Display status of current draft list
            System.out.println("List: " + draftlist);
        }
    }
}
```


Program 10.2 (Run)

```
Run:

Add player:    Input 'a' <name>
Shift player:  Input 's' <from> <to>
Delete player: Input 'r' <name>

List: [Jones, Hardy, Donovan, Bundy]
   Update: a Harrison
List: [Jones, Hardy, Donovan, Bundy, Harrison]
   Update: s 4 2
List: [Jones, Bundy, Hardy, Donovan, Harrison]
   Update: r Donovan
List: [Jones, Bundy, Hardy, Harrison]
   Update: a Garcia
List: [Jones, Bundy, Hardy, Harrison, Garcia]
   Update: s 5 2
List: [Jones, Garcia, Bundy, Hardy, Harrison]
   Update: s 1 4
List: [Garcia, Bundy, Hardy, Jones, Harrison]
   Update: q
```

Palindromes

- A palindrome is a sequence of values that reads the same forward and backward. "level" is a palindrome.
- The method, isPalindrome(), takes a LinkedList object as an argument and returns the boolean value true if the sequence of elements is a palindrome and false otherwise.
- The algorithm compares the elements on opposite ends of the list, using getFirst() and getLast().

isPalindrome()

In the implementation of isPalindrome(), the return type does not depend on a named generic type (return type is boolean). Likewise, the parameter list does not require a named generic type. In this situation, we use a wildcard in the method signature. The syntax

```
LinkedList<?> aList
```

means that aList is a LinkedList object whose elements are of unknown type.

isPalindrome() (concluded)

```
public static boolean isPalindrome(LinkedList<?> aList)
{
   // check values at ends of list as
   // long as list size > 1
   while (aList.size() > 1)
   {
      // compare values on opposite ends; if not equal,
      // return false
      if (aList.getFirst().equals(aList.getLast())
            == false)
         return false;

      // delete the objects
      aList.removeFirst();
      aList.removeLast();
   }

   // if still have not returned, list is a palindrome
   return true;
}
```


Program 10.3

```
import ds.util.LinkedList;
import java.util.Scanner;

public class Program10_3
{
   public static void main(String[] args)
   {
      String str;
      LinkedList<Character> charList =
         new LinkedList<Character>();
      Scanner keyIn = new Scanner(System.in);
      int i;
      char ch;

      // prompt user to enter a string
      // that may include blanks and
      // punctuation marks
      System.out.print("Enter the string: ");
      str = keyIn.nextLine();
```

Program 10.3 (continued)

```
// copy all of the letters as
// lowercase characters to the
// linked list charList
for (i = 0; i < str.length(); i++)
{
   ch = str.charAt(i);

   if (Character.isLetter(ch))
      charList.addLast(Character.toLowerCase(ch));
}

// call isPalindrome() and use return
// value to designate whether the string
// is or is not a palindrome
if (isPalindrome(charList))
   System.out.println("'" + str +
                      "' is a palindrome");
```


Program 10.3 (concluded)

```
      else
         System.out.println("'" + str +
                            "' is not a palindrome");
   }

   < Code for method isPalindrome() >
}
```

```
Run 1:
Enter the string: A man, a plan, a canal, Panama
'A man, a plan, a canal, Panama' is a palindrome

Run 2:
Enter the string: Go hang a salami, I'm a lasagna hog
'Go hang a salami, I'm a lasagna hog' is a palindrome

Run 3:
Enter the string: palindrome
'palindrome' is not a palindrome
```