Data Structures for Java

William H. Ford
William R. Topp

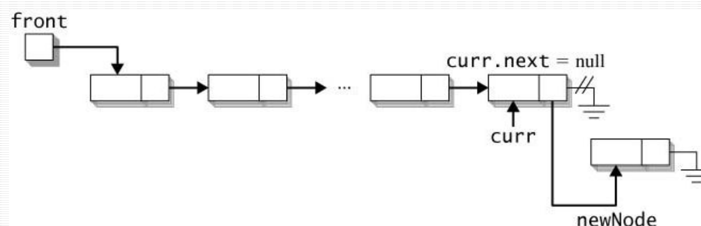
Chapter 11
Implementing the LinkedList Class

Bret Ford

© 2005, Prentice Hall

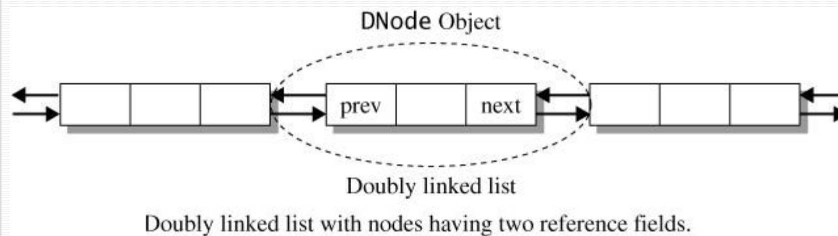
Singly Linked Lists

- Singly-linked lists provide efficient insertion and deletion at the front of the list. Inserting at the back of a singly-linked list is inefficient, because it requires a linear scan that determines the back of the list.



DNode Objects

- A DNode object has reference fields to both the previous node and the next node in the list.
- A sequence of DNode objects creates a list called a *doubly-linked list*.



Doubly linked list with nodes having two reference fields.

DNode Class

- A DNode object has a variable nodeValue that stores the value and two other variables, prev and next, that reference the predecessor and successor of the node, respectively.
- The DNode class is similar to the Node class. It has public data that simplifies access when building implementation structures.
- The class has two constructors. The default constructor creates a DNode object with the nodeValue field set to null. The second constructor provides an argument of type Object to initialize nodeValue.

DNode Class (impl)

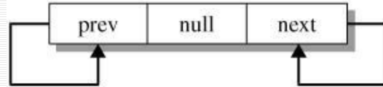
```
public class DNode<T>
{
    public T nodeValue;     // data value of the node
    public DNode<T> prev;  // previous node in the list
    public DNode<T> next;  // next node in the list

    // default constructor; creates an object with
    // the value set to null and whose references
    // point to the node itself
    public DNode()
    {
        nodeValue = null;
        // the next node is the current node
        next = this;
        // the previous node is the current node
        prev = this;
    }
```

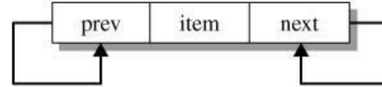
DNode Class (impl)

```
    // creates object whose value is item and
    // whose references point to the node itself
    public DNode(T item)
    {
        nodeValue = item;
        // the next node is the current node
        next = this;
        // the previous node is the current node
        prev = this;
    }
}
```

DNode Class (concluded)

- The declaration of each constructor uses the reference *this* to initialize the links next and prev. The keyword this is a reference to the object itself. The effect is to have each constructor create a node with links that point back to itself.





Creating DNode objects with a null data field or item of generic type T as the data field.

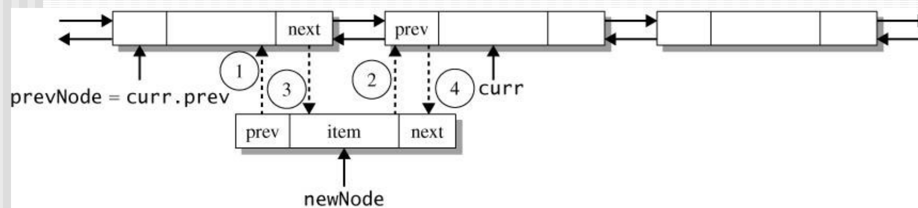
Inserting a Node in DLL

- Inserting a Node At a Position
  - Assume the insertion occurs at reference location curr.
  - Four reference fields in the new node, in node curr, and in node prevNode (curr.prev) must be updated.

Inserting a Node in DLL (2)

```
// declare the DNode reference variables newNode and prevNode
DNode<T> newNode, prevNode;
// create a new node and assign prevNode to reference the
// predecessor of curr
newNode = new DNode<T>(item);
prevNode = curr.prev;

// update reference fields in newNode
newNode.prev = prevNode;   // statement 1
newNode.next = curr;       // statement 2

// update curr and its predecessor to point at newNode
prevNode.next = newNode;   // statement 3
curr.prev = newNode;       // statement 4
```

Inserting a Node in DLL (3)

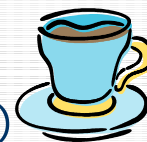


Inserting a node with value item before node curr in a doubly linked list.

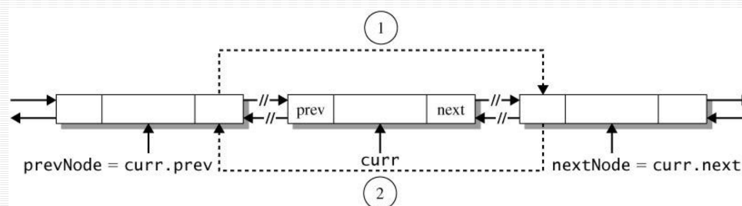
Deleting a Node in a DLL

- Deleting a node at a position.
  - Assume the deletion occurs at reference location curr.
  - The algorithm involves unlinking the node from the list by having the predecessor of curr and the successor of curr point at each other.

Deleting a Node in a DLL (2)



```
DNode<T> prevNode = curr.prev, nextNode = curr.next;
// update the reference variables in the adjacent nodes.
prevNode.next = nextNode;  // statement 1
nextNode.prev = prevNode;  // statement 2
```



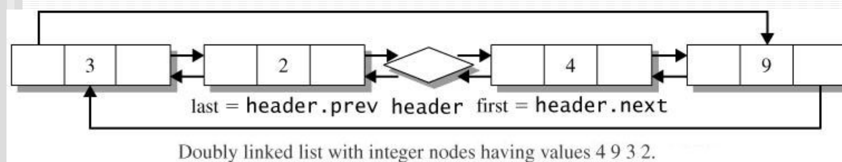
Deleting node curr in a doubly linked list.

Circular Doubly Linked Lists

- A doubly-linked list contains a *sentinel node* called header.
- The sentinel is a DNode object containing a null data value. A linked-list algorithm never uses this value.
- header.next references the first node of the list, and header.prev references the last node.

Circular Doubly Linked Lists (2)

- Traversing a list can begin with the header node and continue either forward or backward until the scan returns to the header.



Doubly linked list with integer nodes having values 4 9 3 2.

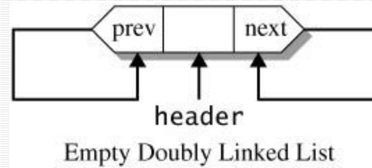
Declaring a DLL

- The declaration of a list begins with the declaration of the header node.
- The default constructor is used to create the header node.

```
DNode<T> header = new DNode<T>();
```

- The declaration of a singly linked list begins by assigning front the value null. The resulting singly-linked list has no nodes.

```
front == null
```



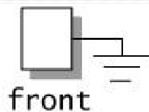
Empty Doubly Linked List

Declaring a DLL (2)

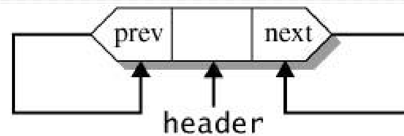
- A doubly-linked list always contains at least one node, the header.

  Empty list:

```
header.next == header
or
header.prev == header
```



Empty Singly Linked List



Empty Doubly Linked List

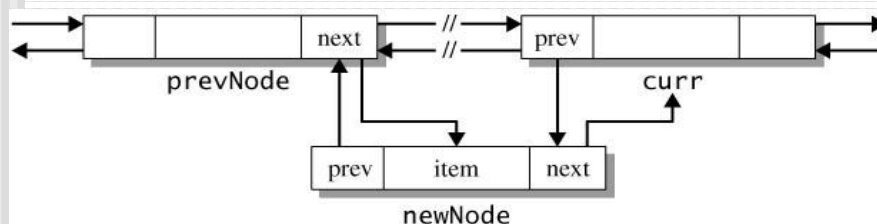
Testing conditions for an empty singly linked list and an empty doubly linked list.

DNodes.toString()

```
public static <T> String toString(DNode<T> header)
{
   if (header.next == header)
      return "null";

 // scan list from first node;add value to string
   DNode<T> curr = header.next;
   String str = "[" + curr.nodeValue;
 // append all but last, separating items with a comma
 // polymorphism calls toString() for the nodeValue type
   while(curr.next != header)
   {
      curr = curr.next;
      str += ", " + curr.nodeValue;
   }
   str += "]";
   return str;
}
```

Inserting a Node

- The insert operation adds a new element at a designated reference location in the list, creates a new node and adds it to the list immediately before the designated node.
- Implemented by `addBefore()` that takes a `DNode` reference argument `curr` and a new value as arguments. The return value is a reference to the new node. The algorithm involves updating four links, so has running time O(1).



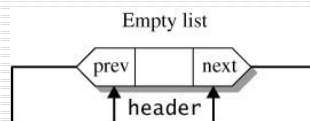
Inserting a Node (2)

```
public static <T> DNode<T> addBefore(DNode<T> curr, T item)
{
  // declare reference variables for new and previous node
   DNode<T> newNode, prevNode;
  // create new DNode with item as initial value
   newNode = new DNode<T>(item);
  // assign prevNode the reference value of node before p
   prevNode = curr.prev;
  // update reference fields in newNode
   newNode.prev = prevNode;
   newNode.next = curr;
  // update curr and prevNode to point at newNode
   prevNode.next = newNode;
   curr.prev = newNode;

   return newNode;
}
```

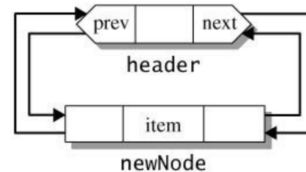
Inserting into an Empty List

- Inserting an element into an empty list simultaneously creates both the first and the last node in the list.
- The header can reference this node by using the `header.next` and the link `header.prev`.

Empty list

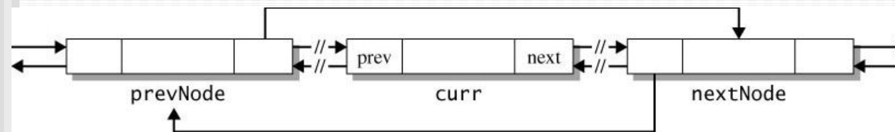


After addBefore(header, item): List with one element



Removing a Node

- The algorithm involves updating links in the adjacent successor and predecessor nodes.
- The method `remove()` takes a `DNode` reference `curr` as an argument. If curr points back to itself (**`curr.next == curr`**), `curr` is the header node of an empty list, and the method simply returns.
- The update of the links requires only two statements, so remove() has running time O(1).



Removing a Node (2)

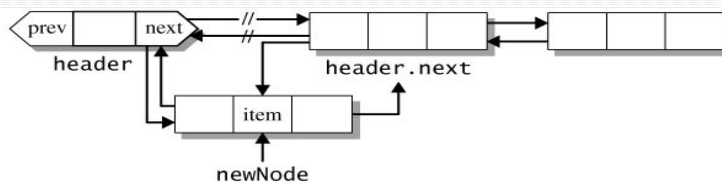
```
public static <T> void remove(DNode<T> curr)
{
    // return if the list is empty
    if (curr.next == curr)
       return;

    // declare references for the predecessor
    // and successor nodes
    DNode<T> prevNode = curr.prev, nexNode = curr.next;

    // update reference fields for
    // predecessor and successor
    prevNode.next = nexNode;
    nexNode.prev = prevNode;
}
```

Modifying the Ends of a List

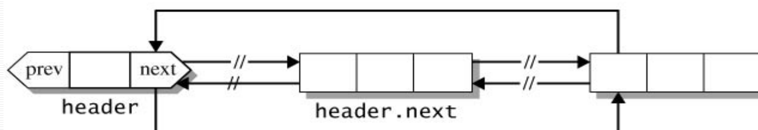
- In a singly-linked list, operations that insert or delete nodes at the ends of the list require very distinct algorithms.
- Because a doubly-linked list is a circular list with a header node, update operations at the ends of the list simply use addBefore() and remove() with arguments that are reference fields in the header.

Modifying the Ends of a List (2)

- To insert item at the front of the list, call `addBefore(header.next, item)`.

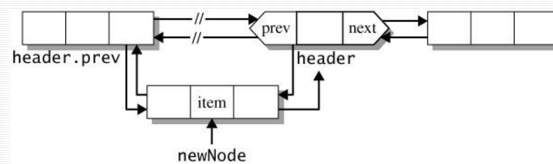


- To remove the front of the list, call `remove(header.next)`.

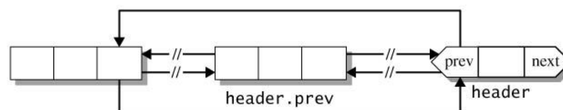


Modifying the Ends of a List (3)

- Update back of list with addBefore() or remove() using header as the argument.



(a) Insert back: addBefore(header, item)



(b) Delete back: remove(header.prev)

jumbleLetters()

```
public static String jumbleLetters(String word)
{
   DNode<Character> header = new DNode<Character>();
   String jumbleword = "";

  // use rnd.nextInt(2) to determine if char is inserted
  // at the front (value = 0) or back (value = 1) of list
   for (int i = 0; i < word.length(); i++)
      if (rnd.nextInt(2) == 0)
             DNodes.addBefore(header.next,word.charAt(i));
      else
             DNodes.addBefore(header, word.charAt(i));
 // create the jumbled word and clear the list
   while (header.next != header)  {
      jumbleword += header.next.nodeValue;
      DNodes.remove(header.next);  }
   return jumbleword;
}
```

Program 11.1

```
import java.util.Random;
import java.util.Scanner;
import ds.util.DNode;
import ds.util.DNodes;

public class Program11_1
{
    static Random rnd = new Random();

    public static void main(String[] args)
    {
        Scanner keyIn = new Scanner(System.in);
        String word, jumbleword;
        int numWords, i, j;
        // prompt for the number of words to enter
        System.out.print("How many words will you" +
                         " enter? ");
```

Program 11.1 (continued)

```
        numWords = keyIn.nextInt();
        for (i = 0; i < numWords; i++)
        {
            System.out.print("Word: ");
            word = keyIn.next();
            jumbleword = jumbleLetters(word);

            // output the word and its jumbled variation
            System.out.println("Word/Jumbled Word: "
                   + word + "    " + jumbleword);
        }
    }
    public static String jumbleLetters(String word)
    {
        DNode<Character> header = new DNode<Character>();
        String jumbleword = "";
```

Program 11.1 (concluded)

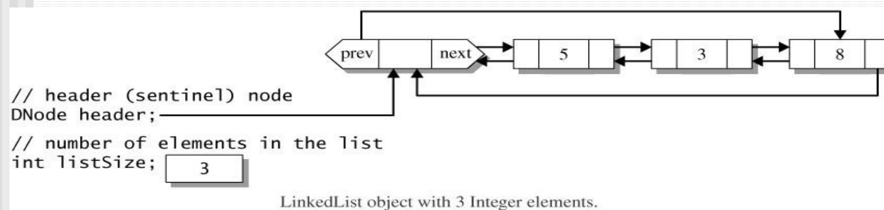
```
        // use rnd.nextInt(2) to determine if char inserted
        // at the front (value = 0) or back (value = 1) of list
        for (int i = 0; i < word.length(); i++)
            if (rnd.nextInt(2) == 0)
                // add at the front of the list
                DNodes.addBefore(header.next,word.charAt(i));
            else
                // insert at the back of the list
                DNodes.addBefore(header, word.charAt(i));
        // create the jumbled word and clear the list
        while (header.next != header)
        {
            jumbleword += header.next.nodeValue;
            DNodes.remove(header.next);
        }
        return jumbleword;
    }
}
```

Program 11.1 (Run)

```
Run:

How many words will you enter? 3
Word: before
Word/Jumbled Word: before   erofeb
Word: java
Word/Jumbled Word: java   vjaa
Word: link
Word/Jumbled Word: link   knli
```

LinkedList Class Private Members

- The DNode reference header identifies the doubly-linked list that stores the elements.
- The integer listSize maintains a count of the number of elements in the list.
- A class method increments modCount whenever its execution updates the list. The variable is used in the implementation of iterators.



LinkedList object with 3 Integer elements.

LinkedList Class Outline

```
public class LinkedList<T> implements List<T>
{
   // number of elements in the list
   private int listSize;
   // the doubly-linked list header node
   private DNode<T> header;
   // maintains a count of the number of list updates
   private int modCount;

   < private utility methods >
   < constructor, List interface, and special purpose methods >
      public LinkedList()
      {
             header = new DNode<T>();
             listSize = 0;
              modCount = 0;
      }
}
```

Index Operations

- For add() and remove() at an index, we need to first determine if the index is in range and then identify the reference that points to the node at the index position. The private methods rangeCheck() and nodeAtIndex() handle these tasks.
- rangeCheck() throws an IndexOutOfBoundsException if an index is not in range.
- nodeAtIndex() returns a DNode reference that points at position index using a loop from 0 to index that tracks a node reference variable which starts at header and moves forward using the next reference field.

Index Operations (2)

```
// return the DNode reference that points at
// an element at position index
private DNode<T> nodeAtIndex(int index)
{
  // check if index is in range
   rangeCheck(index);
  // start at the header
   DNode<T> p = header;
  // go to index either by moving forward from the
  // front of the list
   for (int j = 0; j <= index; j++)
      p = p.next;
  // return reference to node at position p = index
   return p;
}
```

Indexed List Access

- The indexed access methods get() and set() are implemented by using the private method nodeAtIndex().
- These methods are not efficient if they are used repeatedly in a program, because nodeAtIndex() must sequence through the doubly-linked list elements until it reaches the node at position index. A LinkedList collection is really designed to be accessed sequentially.

Indexed List Access (2)

```
// replaces the value at the specified position
// in this list with item and returns the previous value
public T set(int index, T item)
{
   // get the reference that identifies node
   // at position index
   DNode<T> p = nodeAtIndex(index);

   // save the old value
   T previousValue = p.nodeValue;

   // assign item at position index
   p.nodeValue = item;

   // return the previous value
   return previousValue;
}
```

Searching a List

- The method indexOf() takes an argument of type Object and returns the index of the first occurrence of the target in the list or -1 if it is not found. It is used by the index-based add() and remove() methods.
- The related method contains() searches the list and returns a boolean value.

Searching a List (2)

```
public int indexOf(Object item)
{
   int index = 0;
  // search for item using equals()
   for (DNode<T> curr = header.next;
        curr != header; curr = curr.next)
   {
      if (item.equals(curr.nodeValue))  return index;
      index++;
   }
return -1;
}
```

```
public boolean contains(Object item)
{
   return indexOf(item) >= 0;
}
```

Modifying a List

- The LinkedList class has various forms of the add() and remove() methods to insert and delete elements in a list. In each case the method must position itself at the appropriate node. Once the is done, the insertion or deletion operation is executed by calling the corresponding private methods addBefore(curr, item) and remove(curr) where curr is the node reference.

Adding to the Back of a List

- The add(T item) method inserts a new node with value item at the back of the list. Its implementation uses `addBefore(header,item)` with `header` serving as the argument. After updating the list size, add() returns `true`.

```
// appends item to the end of this list and returns true
public boolean add(T item)
{
  // insert item at end of list and increment list size
   DNodes.addBefore(header,item);
   listSize++;
  // the list has changed
   modCount++;
   return true;
}
```

Removing at an Index

- The remove(int i) method deletes the node at position index in the list and returns its value.
- Locating the node is the task of the method call nodeAtIndex() which first uses rangeCheck() to validate the index or throw an IndexOutOfBoundsException. The deletion is handled by the private method remove(curr).

Removing at an Index (2)

```
public T remove(int index)
{
   DNode<T> p = nodeAtIndex(index);

   // save the return value
   T returnElement = p.nodeValue;

   // remove element at node p and decrement list size
   remove(p);
   listSize--;

   // we've made a modification
   modCount++;

   // return the value that was removed
   return returnElement;
}
```