Data Structures for Java

William H. Ford
William R. Topp

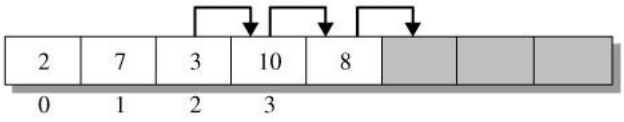
Chapter 10
Linked Lists A

Bret Ford

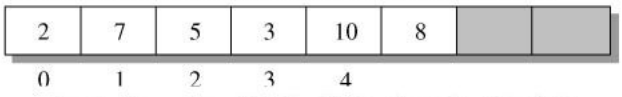
Introducing Linked Lists

- 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 into the ArrayList {2, 7, 3, 10, 8} at the index 2
Make room by shifting the tail {3, 10, 8}



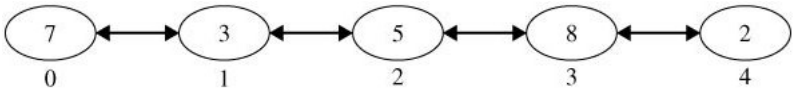
Add 5 at index 2 (Resulting sequence {2, 7, 5, 3, 10, 8})



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

Introducing Linked Lists (2)

- 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 (3)

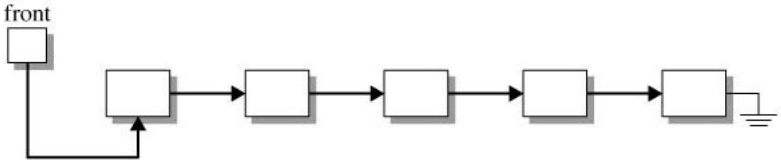
- Inserting and deleting an element is a local operation and requires updating only the links adjacent to the element. The other elements in the list are not affected. An ArrayList must shift all elements on the tail whenever a new element enters or exits the list.

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

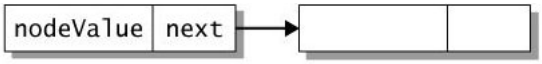
- A singly-linked list is not a direct access structure. It must be accessed sequentially by moving forward one node at a time



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 (2)

- 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

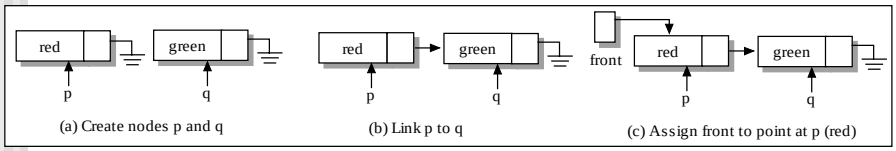
```
public class Node<T>
{
   // data held by the node
   public T nodeValue;
   // next node in the list
   public Node<T> next;
   // default constructor with no initial value
   public Node()
   {
      nodeValue = null;
      next = null;   }
 // initialize nodeValue to item and set next to null
   public Node(T item)
   {
      nodeValue = item;
      next = null;
   }
}
```

Creating a Linked List (3)

- 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.
- 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.

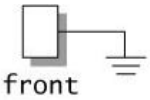
Creating a Linked List (4)

```
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 (5)

- 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()

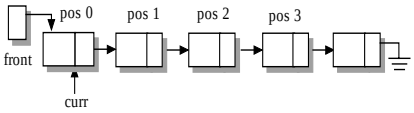
```
public static <T> String toString(Node<T> front)
{
   if (front == null)
      return "null";

   Node<T> curr = front;
   // start with the left bracket and value of first node
   String str = "[" + curr.nodeValue;
  // append all but last node, separating items with a
  // comma
while(curr.next != null)
   {
      curr = curr.next;
      str += ", " + curr.nodeValue;
   }
   str += "]";
   return str;
}
```

Locating a List Position

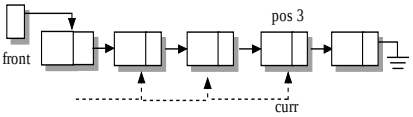
- To locate an element at position n, we need to scan the list through a specified number of node.
- Declare a Node reference curr to point at the first element (front) of the list. This is position 0. A for-loop moves curr down the sequence n times. The variable curr then references the element at position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

Locating a List Position

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



Assign curr to front



Move curr with 3 iterations

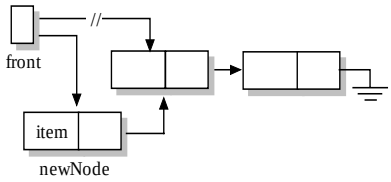
Updating the Front of the List

- 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 (2)

- 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.

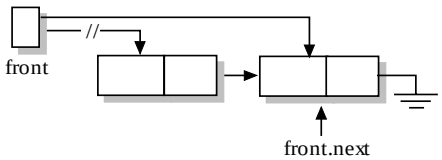
```
Node<T> newNode = new Node<T>(item);

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



Updating the Front of the List (3)

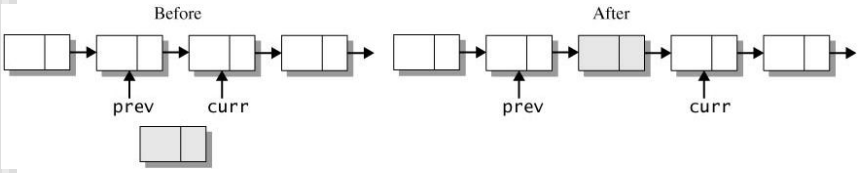
- Deleting the first element involves setting front to reference the second node of the list.

```
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.
- 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.

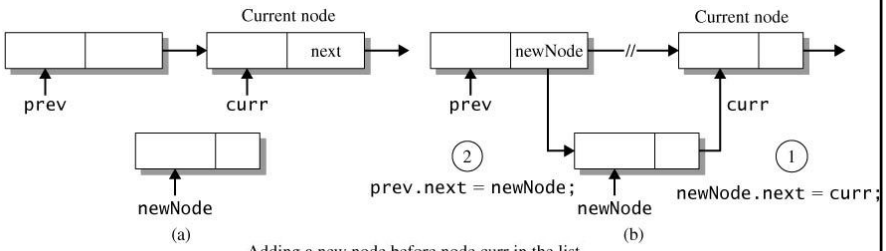


General Insert Operation (2)

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

General Insert Operation (3)

```
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
```



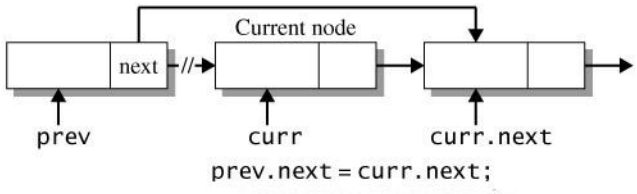
Adding a new node before node curr in the list.

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 (2)

```
Node curr, prev;

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



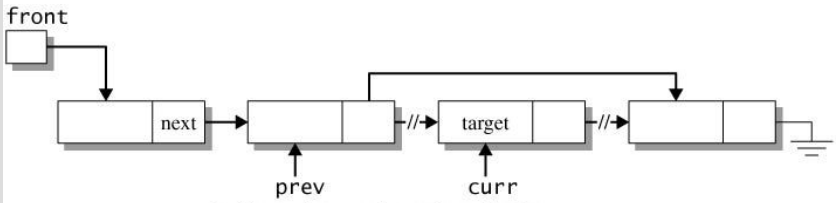
Removing a node at position curr in the list.

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 (2)

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



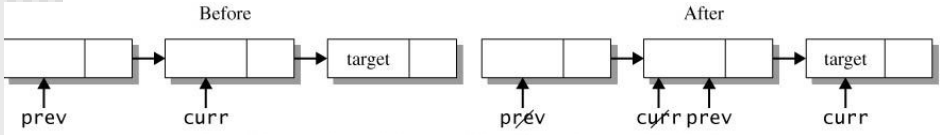
Positions of prev and curr for a deletion.

Removing a Target Node (3)

- 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 (4)



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

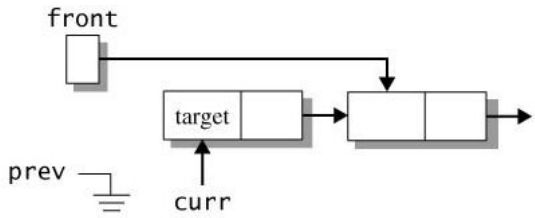
Removing a Target Node (5)

- 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 (6)

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

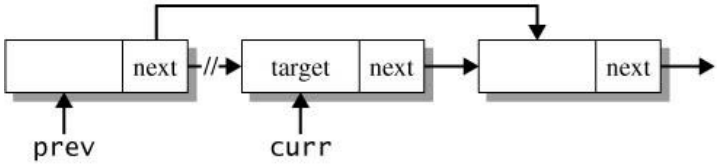
```
front = curr.next;
```



Removing a Target Node (7)

- *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 (8)

- 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 (2)

```
// 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 (3)

```
      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 (2)

```
// 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 (3)

```
        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 (4)

```
// 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 (5)

```
        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
```