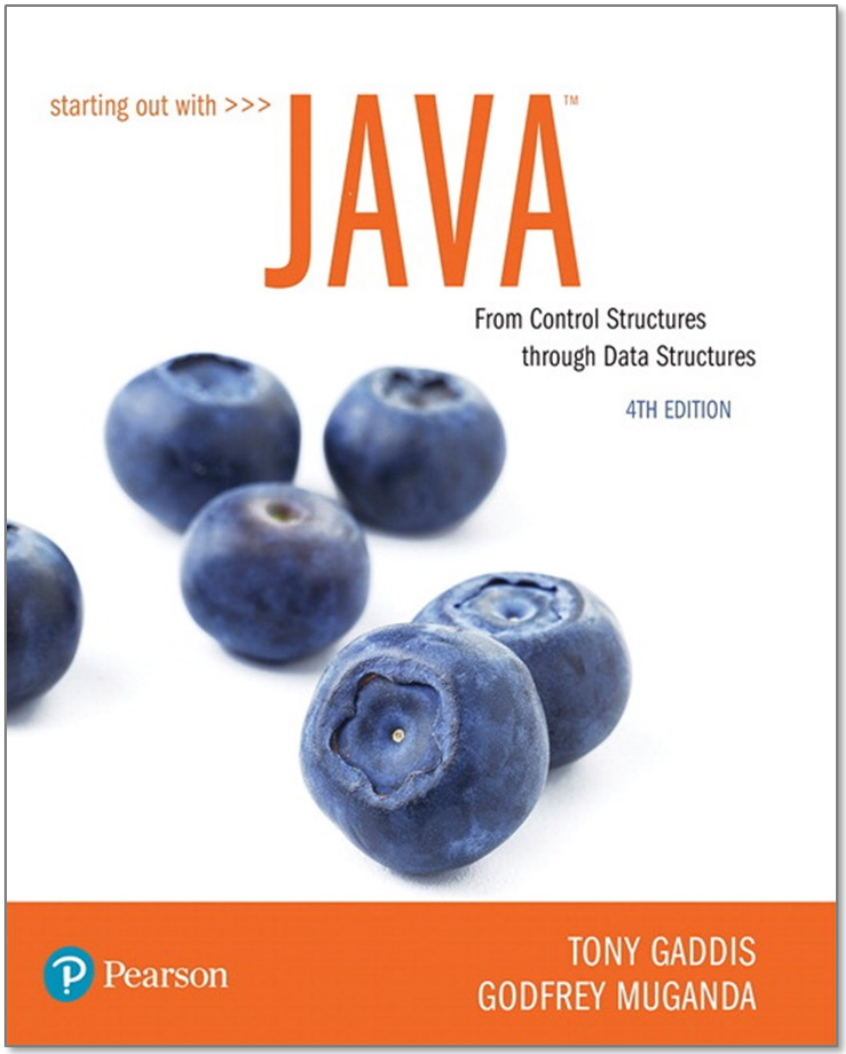
STARTING OUT WITH JAVA™

4th Edition



Chapter 19

Linked Lists

Chapter Topics

- Introduction to Linked Lists
- Linked List Operations
- Doubly Linked and Circularly Linked Lists
- Recursion on Linked Lists

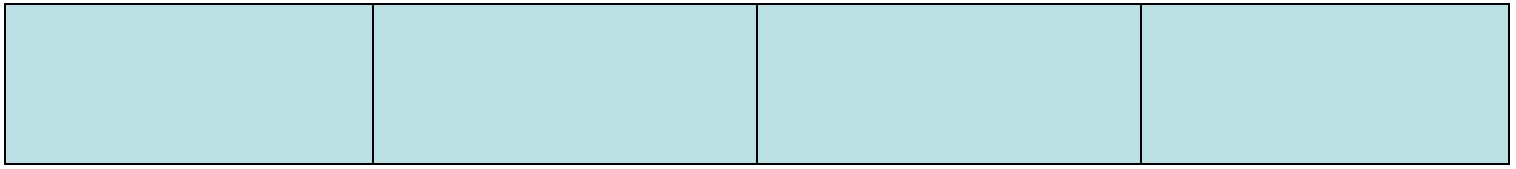
Elementary Facts About Lists

- A list is a collection of objects organized in a sequence.
- Each list element is assigned an integer index based on its position in the sequence.
- The first element in the sequence has index 0, and each succeeding element has index 1 greater than its predecessor.

Contiguous Allocation (1 of 2)

- Implementation of lists can be array-based, this is known as contiguous memory allocation.
- In contiguous allocation, list elements occupy consecutive memory locations.

Contiguous Allocation (2 of 2)



index 0 index 1 index 2 index 3

Random Access

- Access to a contiguously allocated list element is very fast: given any index *k*, we can compute the memory address of the list element at that position by adding the size (in bytes) of *k* items to the address of the first element in the list.

- Contiguous allocation is said to allow random access because we can directly jump to any given element without going through its predecessors.

Contiguous Allocation

Contiguous allocation is used by array-based list implementations such as ArrayList and Vector.

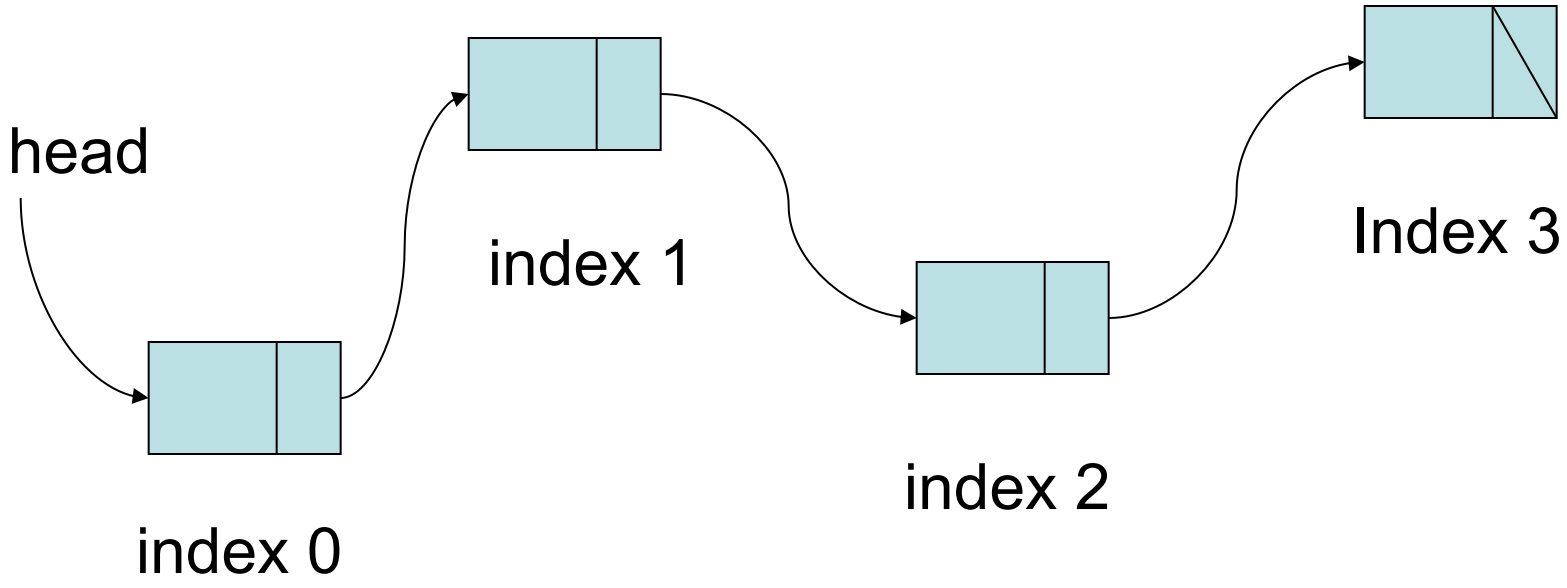
Disadvantages of Contiguous Allocation

- Insertion or deletion of elements in the middle of the list involves the laborious relocation of all elements that come after.
- For example, if a list has a thousand elements, then to insert a new element at position 5 means all elements from position 5 on up must be moved up.
- This overhead is bad for applications that do a lot of insertions and deletions.

Linked Allocation (1 of 2)

- In linked allocation, the list keeps a reference to its first element: this is the head reference
- Every element in the list keeps a reference to its successor, the element that follows it on the list.
- Memory for list elements does not have to be consecutive.

Linked Allocation (2 of 2)



Nodes and Links

- The objects that hold the list element and the reference (link) to its successor are called nodes.

- The references to successors are called successor links.

Disadvantages of Linked Allocation

Given a node in the list, we can only access it by starting at the first node and following the successor links till we come to the desired node.

This is called sequential access.

Sequential access takes a lot longer than random access to get to nodes near the end of a long list.

Java Implementation of Linked Lists

- Requires a Java implementation of a node object.
- A list is represented using a reference to the node with the first element.
- An empty list is represented with a reference whose value is null.
- Each node has a reference to its successor.
- The successor link of the last node is set to null.

Java Implementation of Nodes

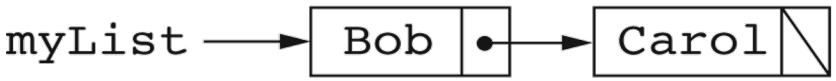
A class that implements a node must hold a list element plus a successor link:

```
//  A node for a list of string elements:
class Node {
   String element;           // list element
   Node next;                // successor link
   Node(String el, Node n) {
      element = el;
      next = n;
   }
   Node(String el) {
      element = el;
      next = null;
   }
}
```

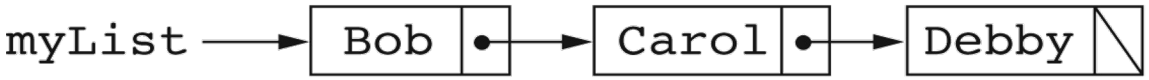
Creation of Linked Lists

```
myList = new Node("Bob");
```

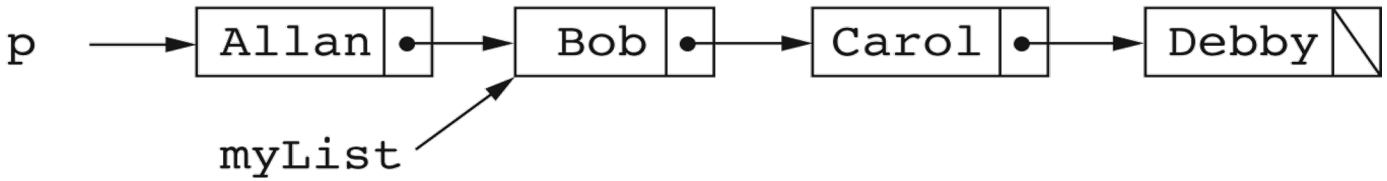

```
myList.next = new Node("Carol");
```




```
myList.next.next = new Node("Debby");
```




```
Node p = new Node("Allan", myList);
```



Inserting Nodes into a Linked List

Say p points to the list [Allan, Bob, Carol, Debby].

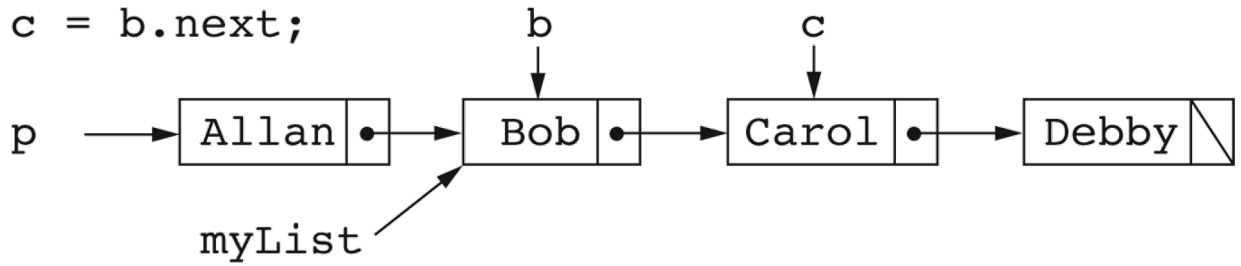
To add “Brad” between “Bob” and “Carol”:

```
Node b = p.next;        //  b points to “Bob”
Node c = b.next;        //  c points to “Carol”
b.next = new Node(“Brad”, c);
```

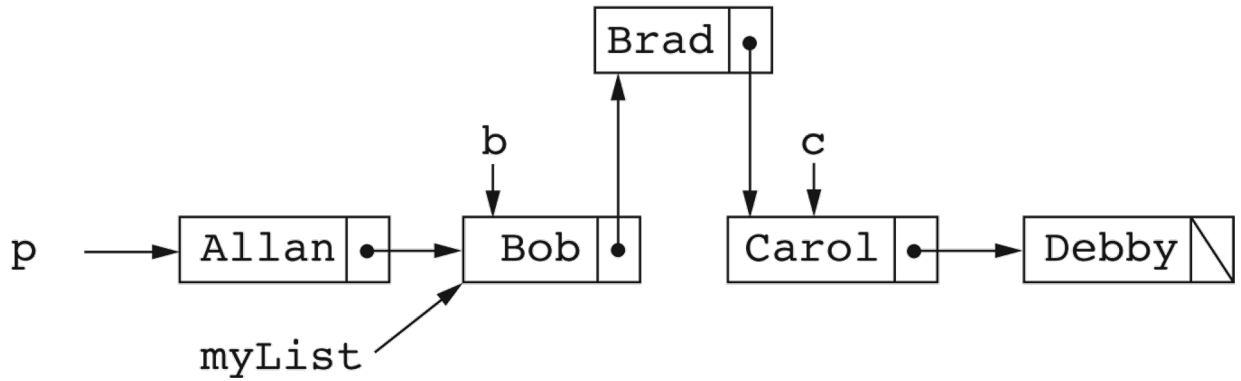
This is illustrated in the next slide.

Figure 2 - Inserting Nodes

```
b = p.next;
c = b.next;
```




```
b.next = new Node("Brad", c);
```



Removing a Node (1 of 2)

To remove the first node of a list, set the head reference to its successor:

```
Node myList = ….;
// Remove first item
myList = myList.next;
```

Removing a Node (2 of 2)

To remove a node other than the first:

1. Set a reference to the predecessor of the targeted node.
2. Route the successor link in the predecessor around the targeted node.

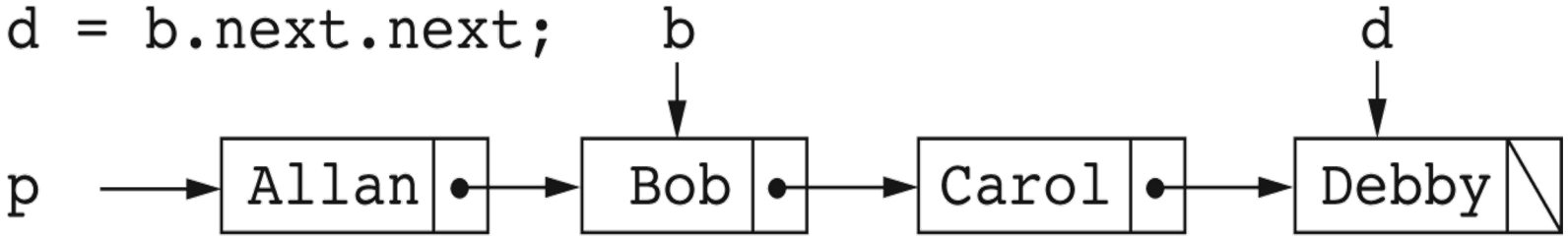
Remove Carol from [Alan Bob Carol Debby]

- Set b to predecessor of Carol (Bob)
- Set d to successor of Carol (Debby)
- Make d the successor of b:

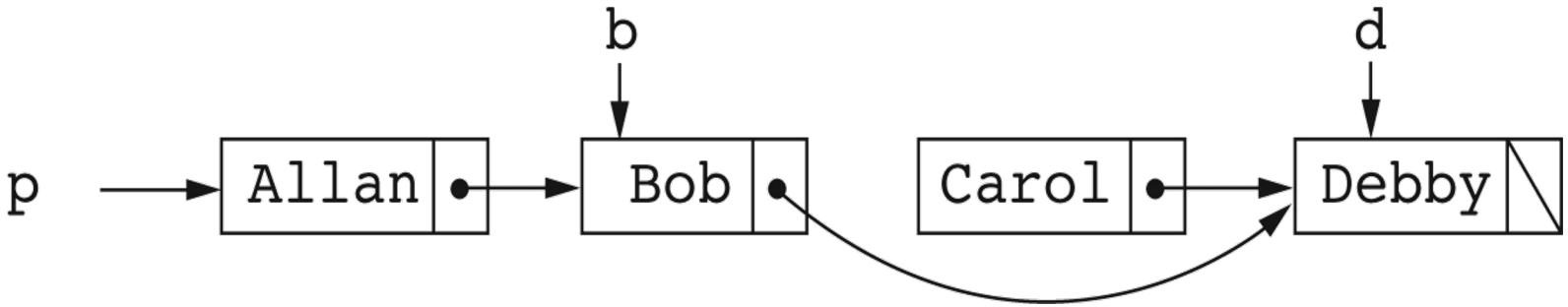
```
b.next = d;
```

Removing a Node

```
b = p.next;
d = b.next.next;
```




```
b.next = d;
```



Traversal of a Linked List (1 of 2)

A traversal of a linked list is a systematic method of examining, in order, every element of the list.

A traversal usually performs some operation as it visits each element.

Traversal of a Linked List (2 of 2)

To traverse a linked list, start a reference at the head of the list, and keep moving it from a node to a successor:

```
Node ref = myList;
while (ref != null) {
    System.out.println(ref.value);
    ref = ref.next;
}
```

Implementation of A Linked List

A linked list class needs

- A node class Node.
- A head reference first to point to the first node:

  Node first;
- A reference last to point to the last node in the list:

  Node last;

The reference last facilitates quick addition of a new node at the end of the list.

Implementation of a Linked List Class

```
class LinkedList1 {
    private class Node {
          String value;
          Node next;
          Node(String val, Node n) {
            value = val;
            next = n;
          }
           Node(String val) {
              // Call the other (sister) constructor.
             this(val, null);
          }
    }
    private Node first;  // list head
    private Node last;   // last element in list
}
```

Class Initialization

The constructor sets first and last to null:

```
public LinkedList1() {
    first = null;
    last = null;
}
```

Checking for an Empty List

A list is empty when first is equal to null:

```
public boolean isEmpty() {
   return first == null;
}
```

Determining the Size of a List

This is done by traversing the list while incrementing a count for each node visited:

```
public int size() {
    int count = 0;
    Node p = first;
    while (p != null) {
        // There is an element at p
        count ++;
        p = p.next;
    }
    return count;
}
```

Adding to the End of the List (1 of 2)

To add e, consider two cases:

1. If list is empty:

```
first = new Node(e);
last = first;
```

2. If list is not empty:

```
last.next = new Node(e);
last = last.next;
```

Adding to the End of the List (2 of 2)

```
public void add(String e) {
    if (isEmpty()) {
        first = new Node(e);
        last = first;
    }
    else {
        // Add to end of existing list
        last.next = new Node(e);
        last = last.next;
    }
}
```

Adding an Element at a Given Index

- Check index: if out of bounds, throw an exception.
- Is index = 0 ? Put new element at beginning of the list, set last and return.
- Is index > 0 ? Skip index nodes from beginning of the list and splice in the new node.
- If the new node was added at the end, set last to the new node.

```
public void add(int index, String e) {
    if (index < 0  || index > size())
        throw new IndexOutOfBoundsException();
    // Index is at least 0
    if (index == 0) {
        // New element goes at beginning
        first = new Node(e, first);
        if (last == null) {last = first;}
        return;
    }
    // Set pred to node that will be the predecessor
    // of the new node
    Node pred = first;
    for (int k = 1; k <= index - 1; k++)
        pred = pred.next;
    // Splice in a node containing the new element
    pred.next = new Node(e, pred.next);
    // Is there a new last element ?
    if (pred.next.next == null)
        last = pred.next;
}
```

Removing a List Element Based on Value

- Use a list traversal to locate the node containing the given value, return false if there is no such node.
- If the target node is the first node, remove it by moving first forward and resetting last if necessary, and then return true.
- Otherwise, find the predecessor of the target node and remove the target node. If the target node was the last node in the list, reset last.

```
public String remove(int index) {
    if (index < 0 || index >= size())
        { throw new IndexOutOfBoundsException(); }
    String element;  // The element to return
    if (index == 0) {
        // Removal of first item in the list
        element = first.element;
        first = first.next;
        if (first == null)  last = null;
        return element;
    }
    // To remove an element other than the first
    // find the predecessor of the element to be removed.
    Node pred = first;0
    // Move pred forward index - 1 times
    for (int k = 1; k <= index -1; k++) { pred = pred.next; }
    // Store the element to return
    element = pred.next.element;
    // Route link around the node to be removed
    pred.next = pred.next.next;
    // Check if pred is now last
    if (pred.next == null)  { last = pred; }
    return element;
}
```

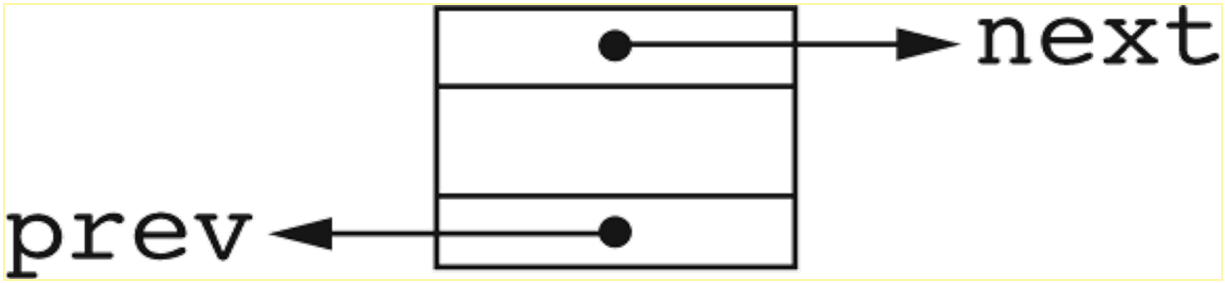
Doubly Linked Lists (1 of 2)

Doubly Linked Lists (2 of 2)

- Singly-linked lists allow one way traversal: one can move from a node to its successor.
- Limitation: one cannot easily move from a node to its predecessor.
- Doubly-linked lists allow easy transitions in both directions.

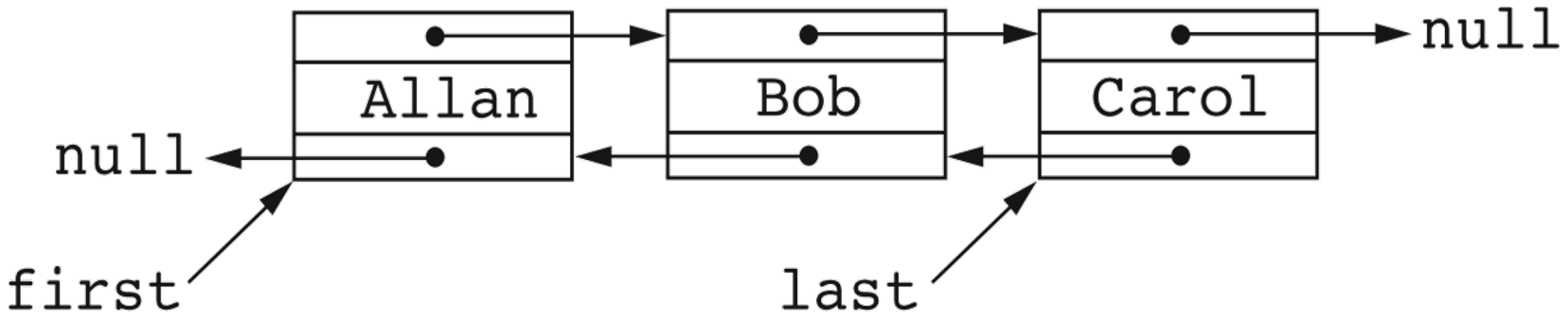
Nodes for Doubly Linked Lists

- A node for a doubly linked list must have both a successor link and a predecessor link:



Doubly-Linked Lists

Can be represented by a head (first) reference and a reference to the last node (for efficiency)



Node Class for Doubly-Linked Lists

```
class Node {
    String element;  // Element
    Node next;       // Reference to successor
    Node prev;       // Reference to predecessor
    Node(String val, Node n, Node p) {
        element = val;
        next = n;
        prev = p;
    }
    Node(String val) {
        // Call the other constructor
        this(val, null, null);
    }
}
```

Operations on Doubly-Linked Lists

- Implementations of size(), isEmpty() are similar to those of singly-linked lists

- Addition and removals are also similar, but require care to manipulate the two links for successor and predecessors.

Addition at beginning of a doubly linked List

To add a new element `e` to an empty list:

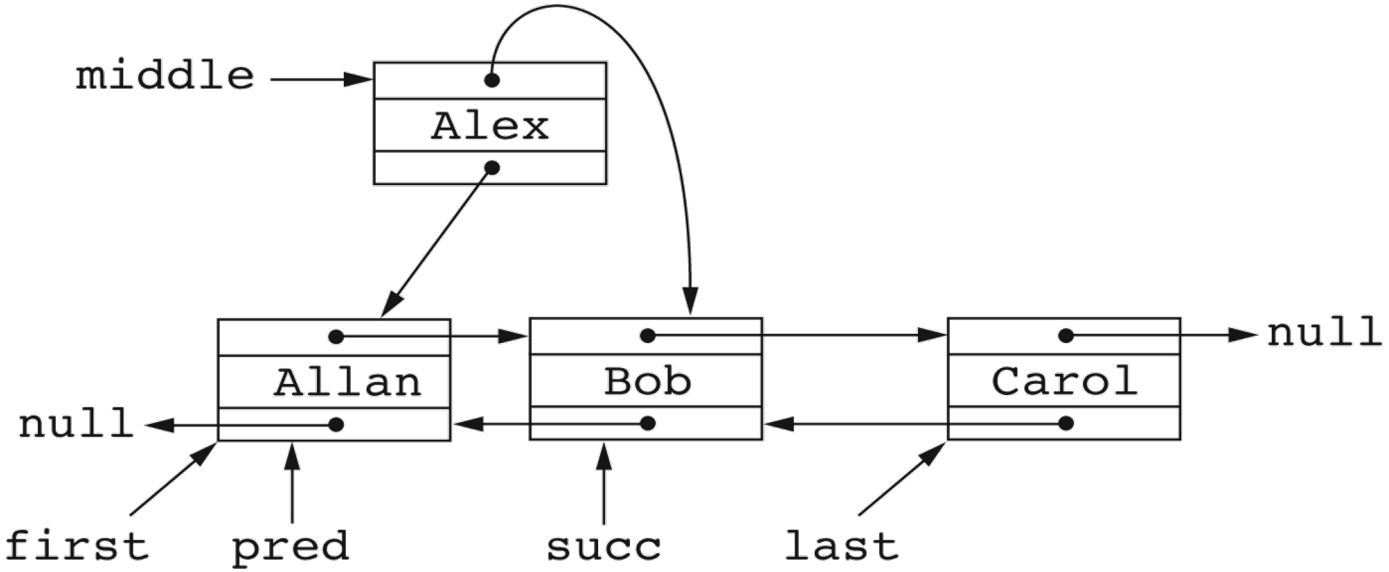
```
first = new Node(e, null, null);
last = first;
```

To add `e` to beginning of a nonempty doubly linked list:

```
first = new Node(e, first, null);
first.next.prev = first;
```

Addition After the First Element

- Set pred to what will be predecessor of the new node.
- Set succ to what will be the successor of the new node.
- Put the new node in the middle (between) pred and succ.



```
public void add(int index, String e) {
    if (index < 0 || index > size()) {
        throw new IndexOutOfBoundsException(); }
    // Index is at least 0
    if (index == 0) {
        // New element goes at beginning
        Node p = first;              // Old first
        first = new Node(e, p, null);
        if (p != null) { p.prev = first; }
        if (last == null) { last = first; }
        return;
    }
    // pred will point to the predecessor of the new node.
    Node pred = first;
    for (int k = 1; k <= index - 1; k++) {
        pred = pred.next;  }
    // Splice in the new node: pred-- succ
    //  becomes  pred--middle--succ
    Node succ = pred.next;
    Node middle = new Node(e, succ, pred);
    pred.next = middle;
    if (succ == null) {
        last = middle; }
    else {
        succ.prev = middle; }
}
```

Recursion on Linked Lists

Recursion on Singly-Linked Lists

- Recursion is natural on lists because a list is a recursive data structure:
  - The empty collection is a list.
  - A non empty list consists of a head (the first element) and a tail.
  - The tail of a list is also a list.

Recursive Implementation of size (1 of 2)

A private int size(Node list) method for computing the size of a list:

- The size of an empty list is 0.
- The size of a non empty list is one greater than the size of the tail: size(list.next) + 1.

Recursive Implementation of size (2 of 2)

```
int size(Node list) {
    if (list == null)
        return 0;
    else
        return size(list.next) + 1;
}
```

Recursive add to End of List

The Node add(String e, Node list) method adds e to the end of list and returns a reference to the first node of the resulting list:

- If list is null, return new Node(e);
- If list is not empty, add e to the end of the tail, attach the original head to the beginning of the modified tail, and return a reference to the original head.

Add e to the End of a List

```
Node add(String e, Node list) {
    if (list == null) {
        return new Node(e);
    }
    else {
        // Replace the tail with result of
        //adding e to the end of the tail

        list.next = add(e, list.next);
        return list;
    }
}
```

Adding an Element at an Index

- In a method

  ```
  Node add(int index, String e, Node list)
  ```

  the value of index must be nonnegative and less or equal to the size of the list.
- The base case is when index is 0: place e at the beginning of the list.
- If index != 0, the list cannot be empty. Place e at position index -1 in the tail of the original list.

Recursive add at an index

```
private Node add(int index, String e, Node list) {
    if (index < 0  || index > size()) {
        String message = String.valueOf(index);
        throw new IndexOutOfBoundsException(message);
    }
    if (index == 0) {
        return new Node(e, list); }

        // 0 < index and index <= size so
        // list is not empty
        // Replace the tail with result
        // of adding e at index - 1 in the tail
        list.next = add(index-1, e, list.next);
        return list;
}
```

Removing a Value From a List

A method for removing needs to return a pair of results:

```
class RemovalResult {
    Node node; // Node containing removed element
    Node list; // List remaining after removal of node
}
```

```
RemovalResult remove(int index, Node list) {
    if (index < 0 || index >= size()) {
        throw new IndexOutOfBoundsException(); }
    if (index == 0) {
        // Remove the first node on list
        RemovalResult remRes =
          new RemovalResult(list, list.next);
       list.next = null;
       return remRes;
    }
    // Recursively remove element at index-1 in the tail
    RemovalResult remRes = remove(index-1, list.next);
    // Replace the tail with the results and return
    // after modifying the list part of RemovalResult
    list.next = remRes.list;
    remRes.list = list;
    return remRes;
}
```

The public Interface to Recursive Methods

In general, recursive methods need to access parameters that refer to private implementation details, and must themselves be private.

Such recursive methods must be called through a public interface that calls a corresponding recursive method.

A Public Interface to size()

```
public int size() {
    // Call the private recursive
    // size(Node p) method
    return size(first);
}
```

Here

```
Node first;
```

is a private reference to the head element of the list.

A Public Interface to remove(int)

This method passes an index to the private method

```
RemovalResult remove(int index, Node List)
```

The public method extracts its return value from the returned RemovalResult object.

Public remove(int)

```
public String remove(int index) {
    // Pass the job on to the recursive version
    RemovalResult  remRes = remove(index, first);

    // Element to return
    String el = remRes.node.value;

    // Remaining list
    first = remRes.list;
    return el;
}
```

Copyright