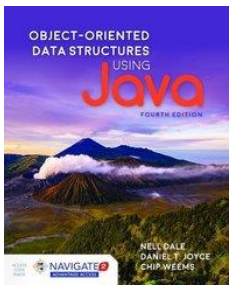
Chapter 6

The
List
ADT



Section 4

LINK-BASED IMPLEMENTATION OF LISTS

Objectives

- Build a single-linked implementation of the list data structure
- Explore advantages and disadvantages the linked-based implementation of lists has over the array-based implementation

3

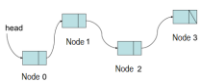
Recall: Links and Nodes

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

4

Linked Allocation

- In linked allocation, the list keeps a reference to its first element: this is the **head** or **firstNode** 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



5

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

6

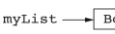
Implementation of Linked Lists

- Requires 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

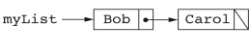
7

Creation of Linked Lists

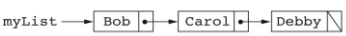
```
myList = new Node("Bob");
```



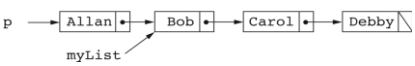

```
myList.next = new Node("Carol");
```




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



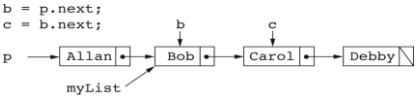

```
Node p = new Node("Allan", myList);
```



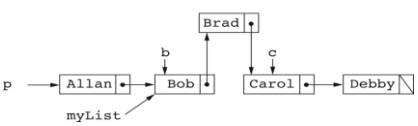
8

Inserting Nodes

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




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



9

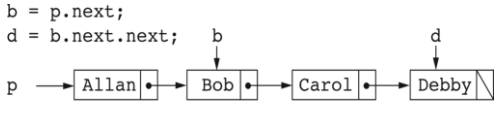
Removing Nodes

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

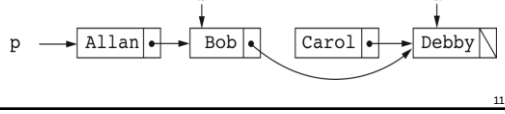
10

Removing Nodes

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




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



11

Traversing a Linked List

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

12

Implementation of a Linked List

- The linked list class can implement the list interface as discussed previously

```
public interface ListInterface<T> {

    int size();
    void reset();
    String toString();
    boolean isEmpty();
    void remove( T target);
    void insert( T element);
    boolean contains( T target);

}
```

13

Implementation of a Linked List

- A linked list class needs
  - A node class: `Node`
  - A reference `firstNode` to point to the first node
  - A reference `lastNode` to point to the last node in the list
- The reference `lastNode` facilitates quick addition of a new node at the end of the list

14

Implementation of a Linked List

```
public class LinkedList<T extends Comparable<T>> implements ListInterface<T> {

    private class Node<T extends Comparable<T>> {

        private T data;
        private Node<T> link;

        public Node( T data) {

            this.data = data;
            this.link = null;
        }
    }

    private Node<T> firstNode;
    private Node<T> lastNode;
```

15

Implementation of a Linked List

- To initialize an abject of the linked list class, the default constructor sets `firstNode` and `lastNode` to `null`
- A copy constructor can also be provided

```
public LinkedList() {

    firstNode = lastNode = null;
}
public LinkedList( LinkedList<T> otherList) {

    if( !otherList.isEmpty()) {

        this.reset();

        T dataValue;
        Node<T> currentNode = otherList.firstNode;

        while( currentNode != null) {
            dataValue = currentNode.data;
            this.insert( dataValue);
            currentNode = currentNode.link;
        }
    }
}
```

16

Implementation of a Linked List

- A list is empty when `firstNode` is equal to `null`
- Determining the size of a list is done by traversing the list while incrementing a `nodeCount` for each node visited

```
public boolean isEmpty() {

    return( firstNode == null);
}
public int size() {

    int nodeCount = 0;
    Node<T> currentNode = firstNode;

    if( isEmpty())
        return nodeCount;

    while( currentNode != null) {
        nodeCount++;
        currentNode = currentNode.link;
    }

    return nodeCount;
}
public void reset() {

    firstNode = lastNode = null;

    return;
}
```

17

Implementation of a Linked List

- Inserting a new node could be at the front of the list, the end of the list, or somewhere in the middle
- We focus on inserting at the end and at the front of a list
- It does not make a difference if the list is empty

```
public boolean insert( T element) {

    Node<T> newNode = new Node<T>( element);

    if( this.isEmpty()) {
        firstNode = lastNode = newNode;
        return true;
    }

    lastNode.link = newNode;
    lastNode = lastNode.link;
    return true;
}
public void insertAtFront( T element) {

    Node<T> newNode = new Node<T>( element);

    if( this.isEmpty()) {
        firstNode = lastNode = newNode;
        return;
    }

    newNode.link = firstNode;
    firstNode = newNode;
    return;
}
```

18

Implementation of a Linked List

- To remove a node, we need to locate it first

```
private Node<T> targetNode;
private Node<T> trailTargetNode;
private boolean found;

private void find( T target) {

    targetNode = firstNode;
    trailTargetNode = null;
    found = false;

    while( targetNode != null && !targetNode.data.equals( target)) {
        trailTargetNode = targetNode;
        targetNode = targetNode.link;
    }

    if( targetNode == null)
        return;
    else
        found = true;

    return;
}
```

```
public boolean remove( T target) {

    find( target);

    if( !found)
        return false;

    if( targetNode == firstNode) {
        firstNode = firstNode.link;
        return true;
    }

    if( size() == 1) {
        reset();
        return true;
    }

    if( targetNode == lastNode) {
        lastNode = trailTargetNode;
        lastNode.link = null;
        return true;
    }

    trailTargetNode.link = targetNode.link;
    return true;
}
```

19

Implementation of a Linked List

- Method `contains` tests if a `target` is in the linked list
- Method `toString` is used to display contents of a linked list

```
public boolean contains( T target) {

    find( target);

    return found;
}

public String toString() {

    String str = "";

    if( isEmpty()) {
        str = "List is empty";
        return str;
    }

    Node<T> current = firstNode;

    while( current != null) {
        str = str + current.data + "\t";
        current = current.link;
    }

    return str;
}
```

20

Testing the `LinkedList` Class

```
public class LListDriver {

    public static void main(String[] args) {

        LinkedList<String> names = new LinkedList<String>();

        names.insert( "Bob");
        names.insert( "Mary");
        names.insert( "Mike");
        System.out.println( names);
        System.out.println( "There are " + names.size() + " names in names.");
        names.insertAtFront( "Smith");
        System.out.println( names);
        System.out.println( "There are " + names.size() + " names in names.");
        names.remove( "Bob");
        System.out.println( names);
        System.out.println( "There are " + names.size() + " names in names.");
        names.remove( "Smith");
        System.out.println( names);
        System.out.println( "There are " + names.size() + " names in names.");
        names.remove( "Mike");
        System.out.println( names);
        System.out.println( "There are " + names.size() + " names in names.");
        names.remove( "Mary");
        System.out.println( names);
        System.out.println( "There are " + names.size() + " names in names.");
    }
}
```

21