

Data Structures and Algorithms

Chapter 3

Linked Lists

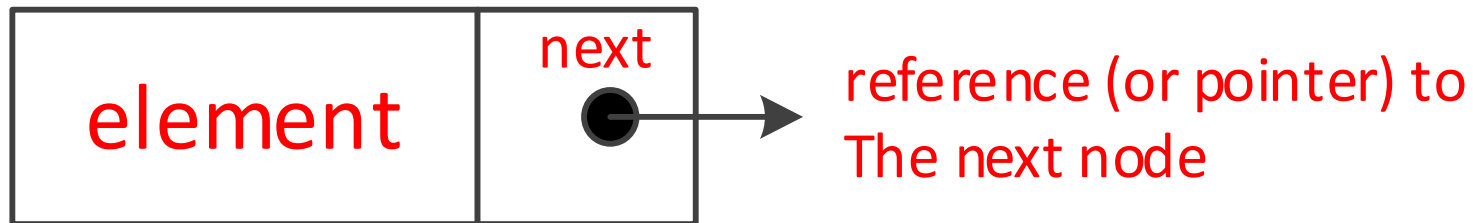
- A node stores an element and a link (or links).
- Nodes are connected by links.
- A link is the reference to (or the address of) a node.
- Singly linked list, doubly linked list, circularly linked list
- *Link*, *reference*, and *pointer* are used interchangeably.

Linked Lists

- Need to learn
 - Different linked data structures
 - To write code creating and manipulating linked data structures.
 - To use predefined linked data structures, such as Java's *LinkedList*

Singly Linked Lists

- Nodes are connected by a single link.
- A link points to (or references) the next node.
- A node has *element* and the reference (or pointer) to the next node.

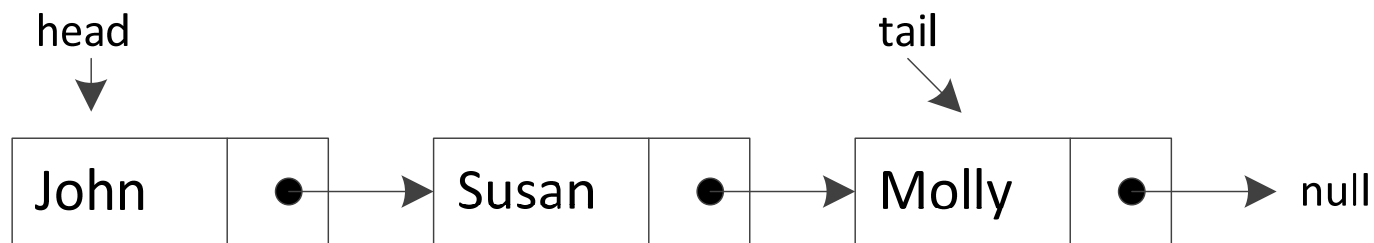


Singly Linked Lists

```
private static class Node<E> {  
    private E element;  
    private Node<E> next;  
    public Node(E e, Node<E> n) {  
        element = e;  
        next = n;  
    }  
    public E getElement() { return element; }  
    public Node<E> getNext() { return next; }  
    public void setNext(Node<E> n) { next = n; }  
}
```

Singly Linked Lists

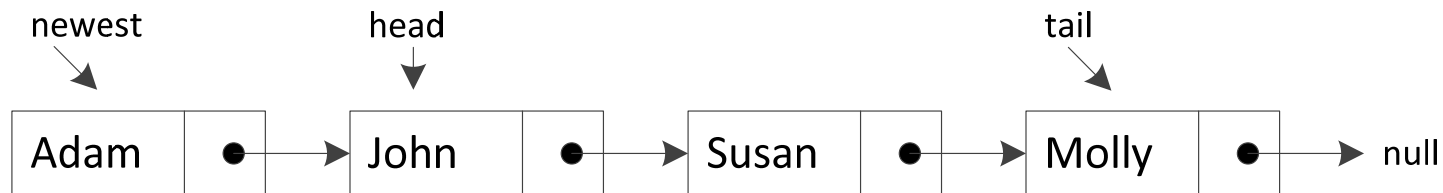
- We usually keep two references, *head* and *tail*, for a singly linked list.



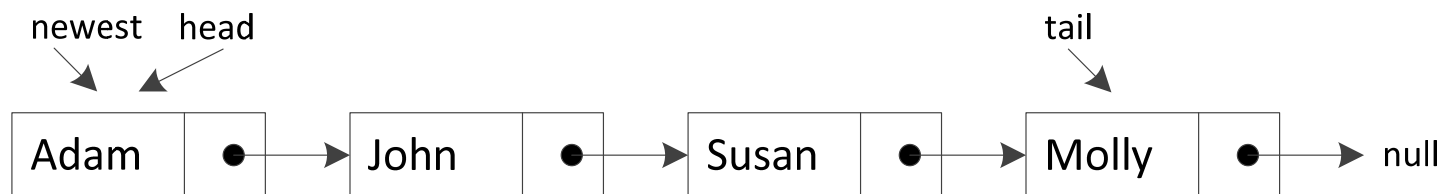
Singly Linked Lists

- Add a node to the head of a list

```
newest = Node("Adam"); newest.next = head;
```



```
head = newest;
```



Singly Linked Lists

- Add a node to the head of a list

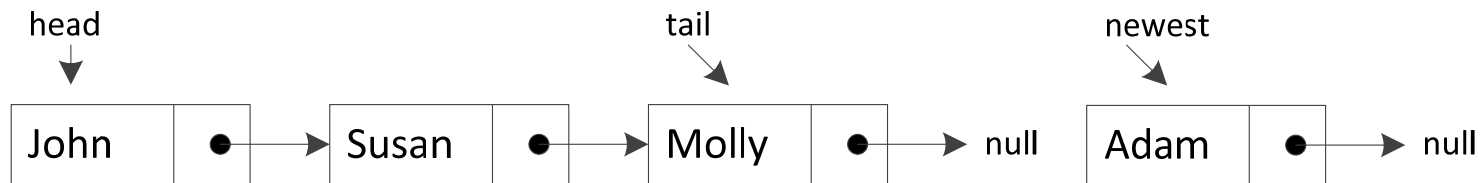
Algorithm addFirst(e)

```
newest = Node(e) // new node with element e
newest.next = head // new node's next is set to refer
                  // to current head node
head = newest // new head refers to new node
size = size + 1 // list size (node count) is
               // incremented
```

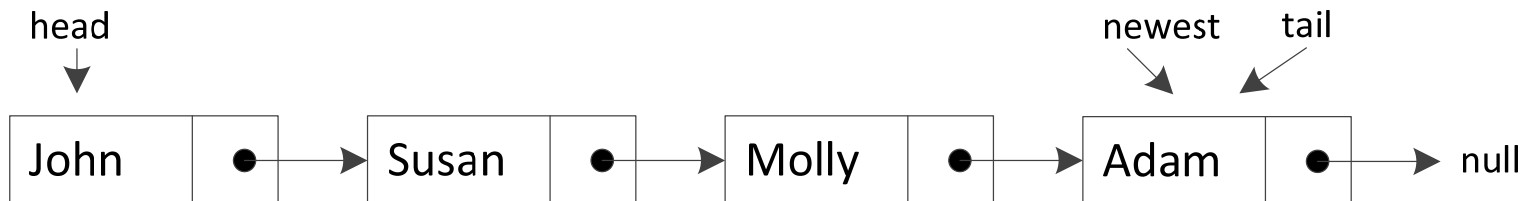

Singly Linked Lists

- Add a node to the tail of a list

```
newest = Node("Adam"); newest.next = null ;
```



```
tail.next = newest; tail = newest;
```



Singly Linked Lists

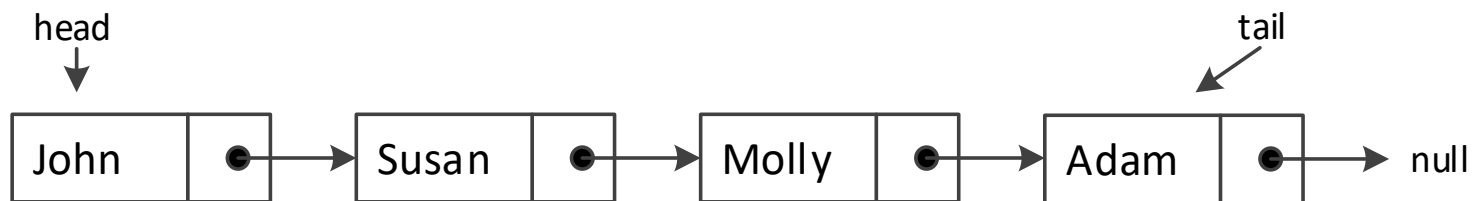
- Add a node to the tail of a list

Algorithm addLast(*e*)

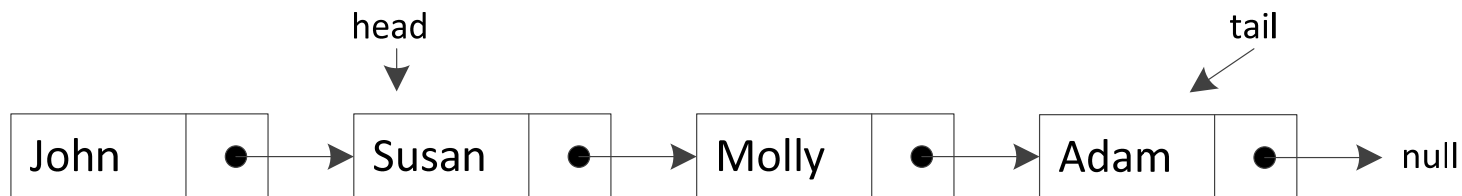
```
newest = Node(e) // new node with element e
newest.next = null // new node's next is set to null
tail.next = newest // current tail node's next points to
                  // new node
tail = newest      // new tail points to new node
size = size + 1   // list size (node count) is
                  // incremented
```

Singly Linked Lists

- Removing an arbitrary node: nontrivial and inefficient.
- Removing a node from the head of a list



`head = head.next;`



Singly Linked Lists

Algorithm removeFirst()

if head == null

the list is empty

head = head.next // new head points to next node

size = size - 1 // list size (node count) is
// decremented

Singly Linked Lists

- Generic *Node* class in *SinglyLinkedList* class

```
1 private static class Node<E> {  
2     private E element;  
3     private Node<E> next;  
4     public Node(E e, Node<E> n) {  
5         element = e;  
6         next = n;  
7     }  
8     public E getElement() { return element; }  
9     public Node<E> getNext() { return next; }  
10    public void setNext(Node<E> n) { next = n; }  
11 }
```

Singly Linked Lists

- Instance variables of *SinglyLinkedList* class
public class SinglyLinkedList<E> implements Cloneable
{
 // nested class Node
 protected Node<E> head = null;
 protected Node<E> tail = null;
 protected int size = 0;
 // constructors and methods

Singly Linked Lists

- Complete code of *SinglyLinkedList.java*

Doubly Linked Lists

- Singly linked list:
 - Not easy to insert a node at an arbitrary position.
 - Nontrivial to delete a node from an arbitrary position.
- These operations, however, can be performed relatively efficiently with doubly linked lists.

Doubly Linked Lists

- A node has two pointers.
- *previous* points to the previous node.
- *next* points to the next node.

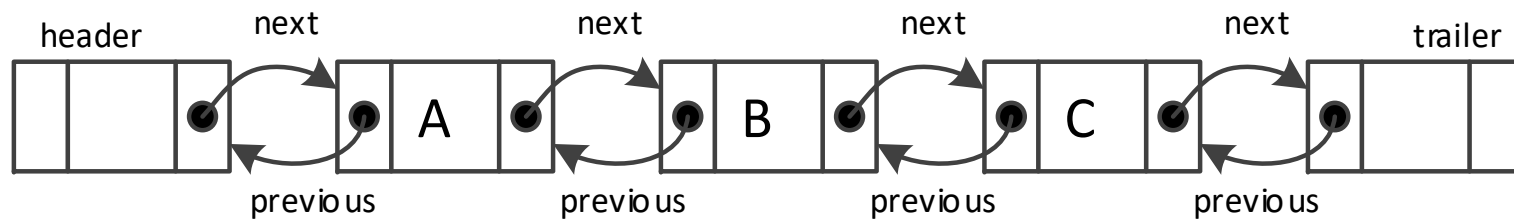


Doubly Linked Lists

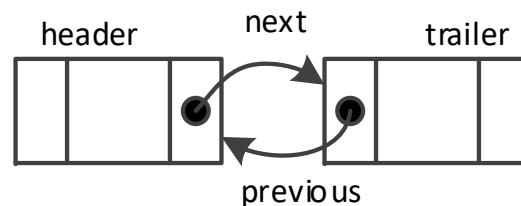
```
private static class Node<E> {  
    private E element;  
    private Node<E> prev;  
    private Node<E> next;  
    public Node(E e, Node<E> p, Node<E> n) {  
        element = e; prev = p; next = n;  
    }  
    public E getElement() { return element; }  
    public Node<E> getPrev() { return prev; }  
    public Node<E> getNext() { return next; }  
    public void setPrev(Node<E> p) { prev = p; }  
    public void setNext(Node<E> n) { next = n; }  
}
```

Doubly Linked Lists

- Doubly linked list example

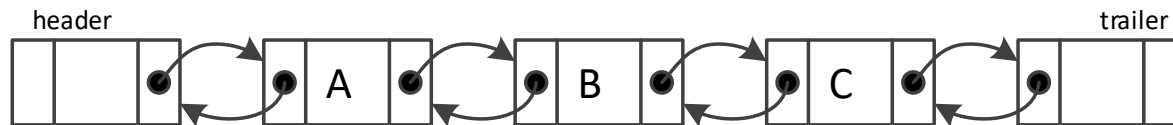
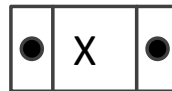


- An empty list

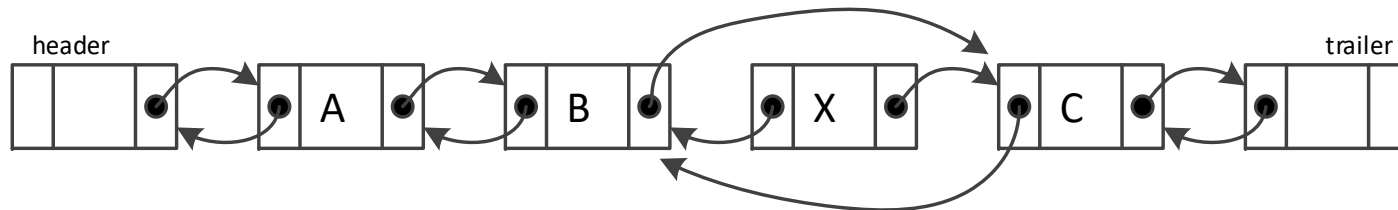


Doubly Linked Lists

- Insert a new node X between B and C

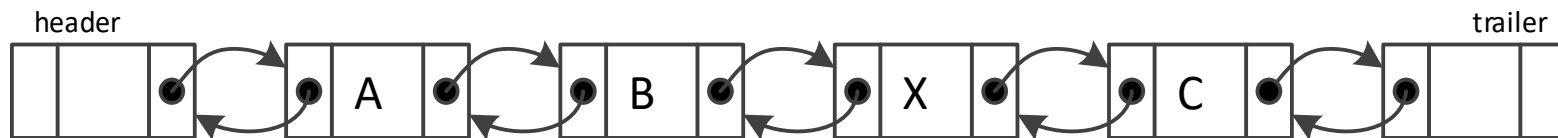


- The previous reference of X are set to point to B.
- The next reference of X are set to point to C.



Doubly Linked Lists

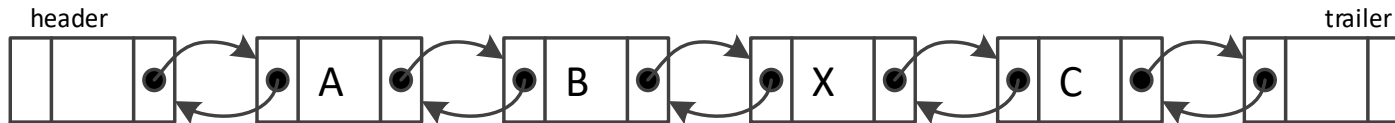
- The next reference of B and the previous reference of C are updated to point to X.



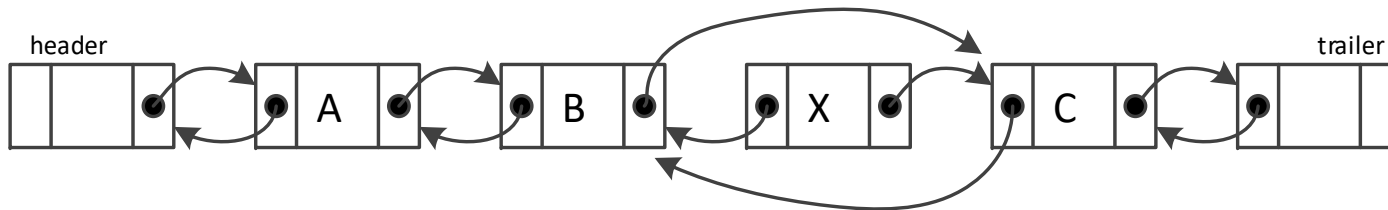
```
1  private void addBetween(E e, Node<E> predecessor,
                                Node<E> successor) {
2  Node<E> newest = new Node<>(e, predecessor, successor);
3  predecessor.setNext(newest);
4  successor.setPrev(newest);
5  size++;
6  }
```

Doubly Linked Lists

- Delete X

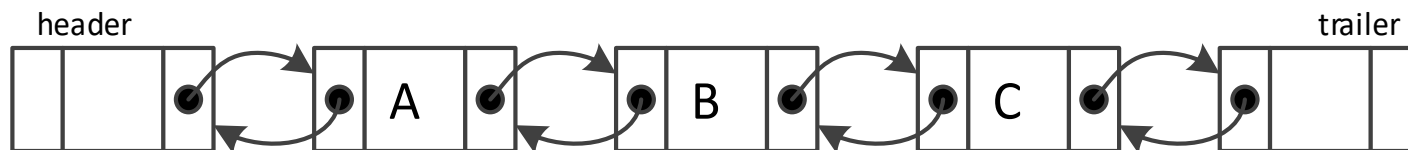


- Set the next reference of B to point to C
- Set the previous reference of C to point to B.



Doubly Linked Lists

- X is not a part of the list any more. The updated list is:



```
1 private E remove(Node<E> node) {  
2     Node<E> predecessor = node.getPrev();  
3     Node<E> successor = node.getNext();  
4     predecessor.setNext(successor);  
5     successor.setPrev(predecessor);  
6     size--;  
7     return node.getElement();  
8 }
```

Doubly Linked Lists

- A complete code of *DoublyLinkedList.java*

Insertion Sort

- There are different sorting algorithms.
- Will discuss insertion sort algorithm (on an array).
- Pseudocode

Algorithm InsertionSort(A)

Input: Array A of n comparable elements

Output: Array A with elements rearranged in
nondecreasing order

for k from 1 to $n - 1$ **do**

 Insert $A[k]$ at its proper location within $A[0 .. k]$

Insertion Sort

- Illustration

k = 1

17	6	12	32	24	8	14	11
0	1	2	3	4	5	6	7

k = 2

6	17	12	32	24	8	14	11
0	1	2	3	4	5	6	7

k = 3

6	12	17	32	24	8	14	11
0	1	2	3	4	5	6	7

k = 4

6	12	17	32	24	8	14	11
0	1	2	3	4	5	6	7

k = 5

6	12	17	24	32	8	14	11
0	1	2	3	4	5	6	7

k = 6

6	8	12	17	24	32	14	11
0	1	2	3	4	5	6	7

k = 7

6	8	12	14	17	24	32	11
0	1	2	3	4	5	6	7

6	8	11	12	14	17	24	32
0	1	2	3	4	5	6	7

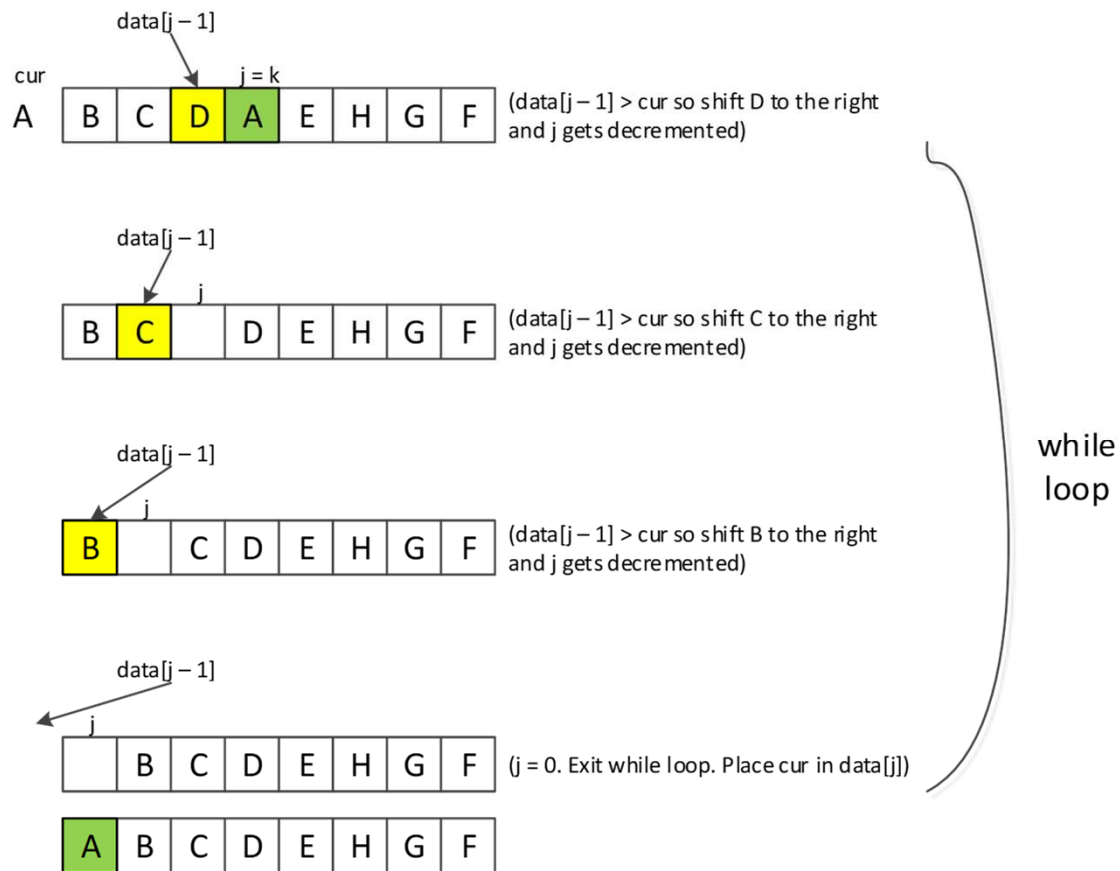
Insertion Sort

- Java implementation

```
1 public class InsertionSort {
2     public static void insertionSort(char[ ] data) {
3         int n = data.length;
4         for (int k = 1; k < n; k++) { // begin with second element
5             char cur = data[k];      // save data[k] in cur
6             int j = k;                // find correct index j for cur
7             while (j > 0 && data[j-1] > cur) { // thus, data[j-1] must go after cur
8                 data[j] = data[j-1];    // shift data[j-1] rightward
9                 j--;                    // and consider previous j for cur
10            } // while
11            data[j] = cur;                // this is the proper place for cur
12        } // for
13    } // running time: O(n2)
```

Insertion Sort

- Illustration of while loop in Java implementation



Testing Equality

- When comparing two reference variables, there are two notions of equivalence.
- First interpretation: Test whether two reference variables are pointing to the same object.
- Second interpretation: Test whether the contents of the two objects pointed to by the references are the same.

```
String s1 = new String("data structure");
```

```
String s2 = new String("data structure");
```

- Is s1 equal to s2?
 - No, by the first interpretation
 - Yes, by the second interpretation

Testing Equality

- In Java, you can compare with “==” operator or using the *equals* method.
- “==” compares the values of the reference variables, i.e., it checks whether they refer to the same object.
- The *equals* method is defined in the *Object* class, and, as it is, it is effectively the same as “==” operator.
- To implement the “second interpretation” for objects of a class, the class must define its own *equals* method tailored for the objects of that class.

Testing Equality

- String class has *equals* method which performs character-by-character, pair-wise comparison.

```
1 public class StringTest {  
2     public static void main(String[] args) {  
3         String s1 = new String("data structure");  
4         String s2 = s1;  
5         String s3 = new String("data structure");  
6         System.out.println("reference s1 equals reference s2: " + (s1 == s2));  
7         System.out.println("reference s1 equals reference s3: " + (s1 == s3));  
8         System.out.println("string s1 equals string s3: " + s1.equals(s3));  
10    }  
11 }
```

- Output: true, false, true

Testing Equality

- Equivalence testing with arrays
 - `a == b`: Tests if `a` and `b` refer to the same array instance.
 - `a.equals(b)`: This is identical to `a == b`.
 - `Arrays.equals(a, b)`: Returns true if the arrays have the same number of elements and all pairs of corresponding elements are equal to each other. If elements are primitives, `==` operator is used. If elements are reference types, then `a[k].equals(b[k])` is used.

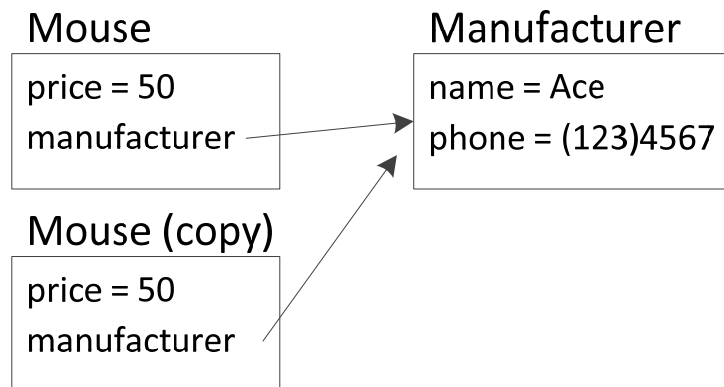
Testing Equality

- Equivalence testing with linked lists
 - Traverse two lists and compare pairs of corresponding elements.
 - Refer to the *equals* method in the *SinglyLinkedList* class.

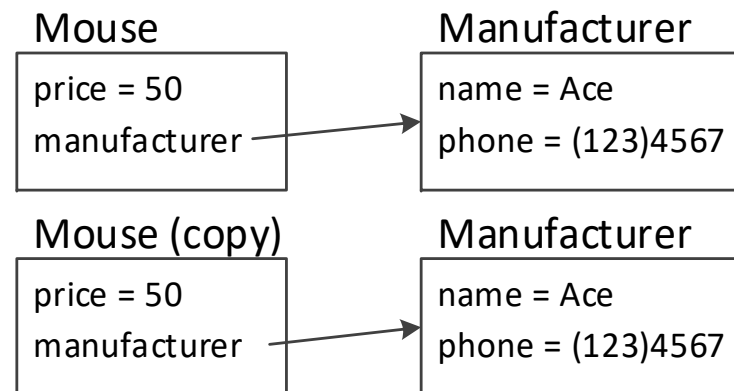
Cloning Data Structures

- Shallow copy vs. deep copy

Shallow copy



Deep copy



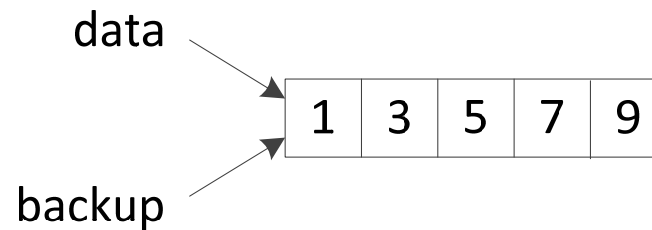
Cloning Data Structures

- Java's Object class has the *clone* method.
- This clone method creates a shallow copy.
- If necessary, each class must define its own clone method.

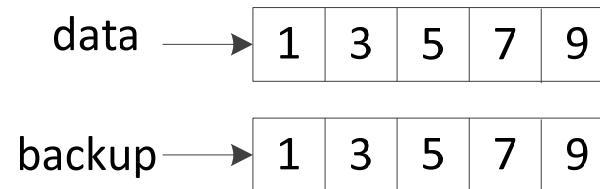
Cloning Data Structures

- Cloning arrays with elements of primitive type

```
int[ ] data = {1,3,5,7,9};  
int[ ] backup;  
backup = data;
```



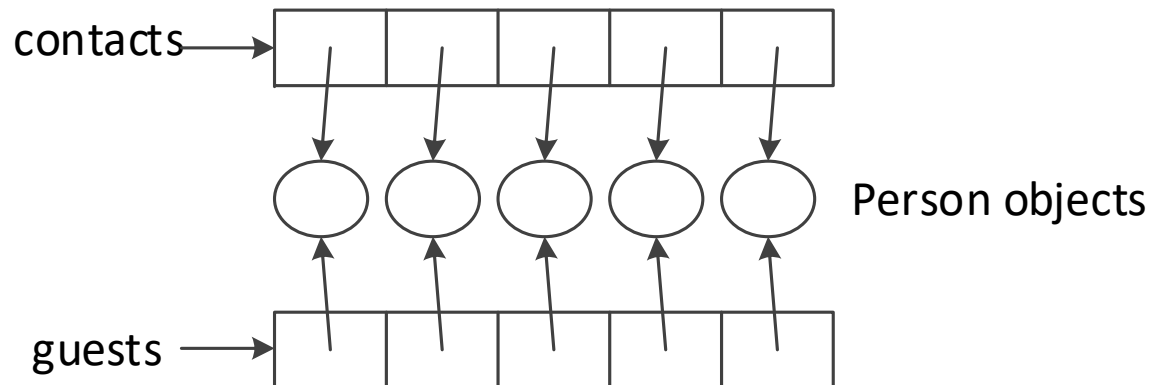
```
backup = data.clone();
```



Cloning Data Structures

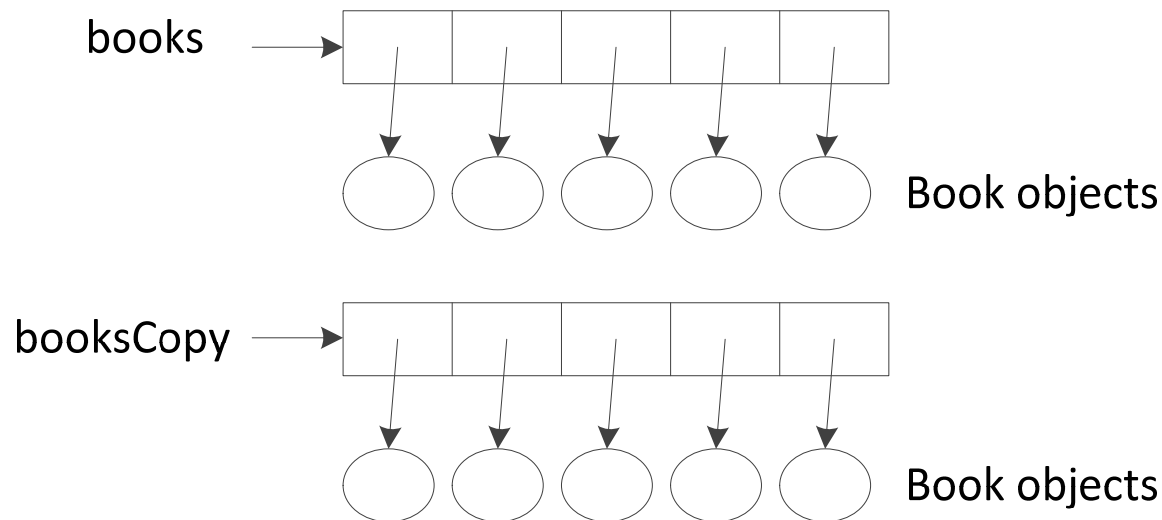
- Cloning arrays with elements of object type

```
guests = contacts.clone( ); // a shallow copy is created
```



Cloning Data Structures

- Cloning arrays with elements of object type
 - The following is a deep copy.
 - A separate code must be written.



Cloning Data Structures

- Cloning linked lists:
 - Must copy one node at a time.
 - Refer to the *clone* method in *SinglyLinkedList* class.

References

- M.T. Goodrich, R. Tamassia, and M.H. Goldwasser, “Data Structures and Algorithms in Java,” Sixth Edition, Wiley, 2014.