Preliminaries

- Options for implementing a list ADT
  - Array
    - Has a fixed size
    - Data must be shifted during insertions and deletions
  - Linked list
    - Is able to grow in size as needed
    - Does not require the shifting of items during insertions and deletions

An Array-Based Implementation of the ADT List

- An array-based implementation
  - A list's items are stored in an array `items`
  - A natural choice
    - Both an array and a list identify their items by number
  - A list's k^{th} item will be stored in `items[k-1]`

An Array-Based Implementation of the ADT List

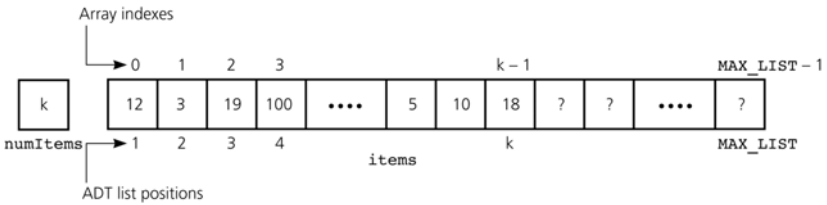


Figure 4-11

An array-based implementation of the ADT list

Linked list preliminaries

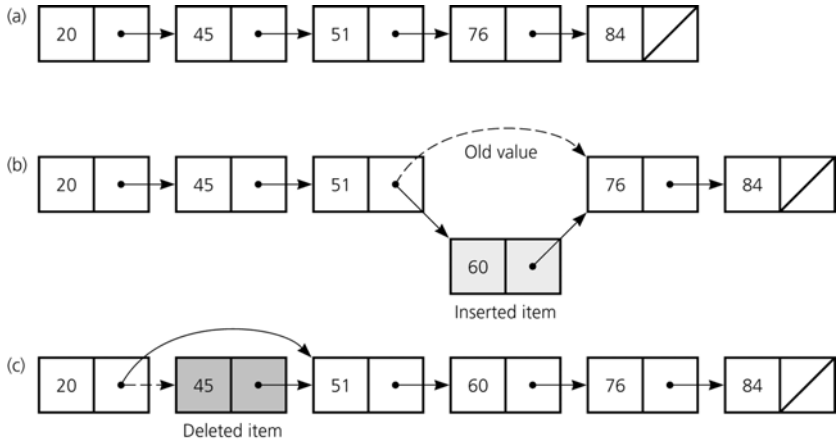


Figure 5-1

a) A linked list of integers; b) insertion; c) deletion

Object References

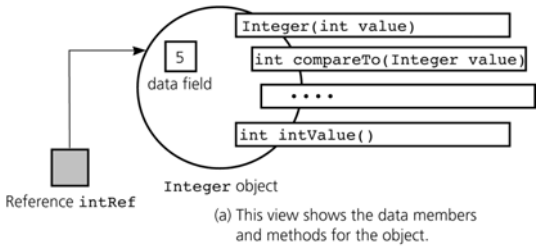
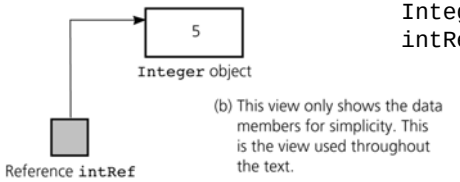


Figure 5-2
A reference to an ***Integer*** object




```
Integer intRef;
intRef = new Integer(5);
```


Object References

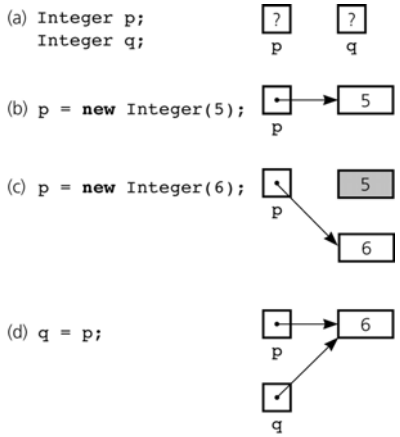


Figure 5-3a-d
a) Declaring reference variables; b) allocating an object; c) allocating another object, with the dereferenced object marked for garbage collection

Object References

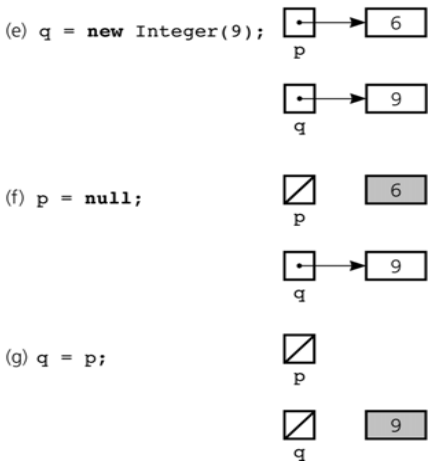


Figure 5-3e-g
e) allocating an object; f) assigning ***null*** to a reference variable; g) assigning a reference with a ***null*** value

Reference-Based Linked Lists

- Linked list
  - Contains nodes that are linked to one another
  - A node
    - Contains both data and a "link" to the next item
    - Can be implemented as an object

```
public class Node {
    private Object item;
    private Node next;

    // constructors
    // and other methods
}
```

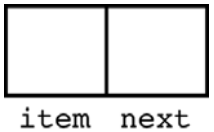


Figure 5-5
A node

Reference-Based Linked Lists

- Using the Node class

```
Node n = new Node (new Integer(6));
Node first = new Node (new Integer(9), n);
```

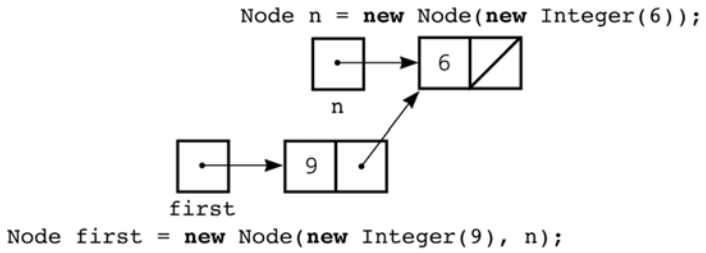


Figure 5-7

Using the ***Node*** constructor to initialize a data field and a link value

Node Constructors

```
private Object item;
private Node next;

public Node(Object item) {
   this.item = item;
   next = null;
}

public Node(Object item, Node next) {
   this.item = item;
   this.next = next;
}
```


Node Methods

```
public Object getItem() {
   return item;
}

public void setItem(Object item) {
   this.item = item;
}

public Node getNext() {
   return next;
}

public void setNext(Node next) {
   this.next = next;
}
```


Reference-Based Linked Lists

- Data field `next` in the last node is set to `null`
- `head` reference variable
  - References the list's first node
  - Always exists even when the list is empty

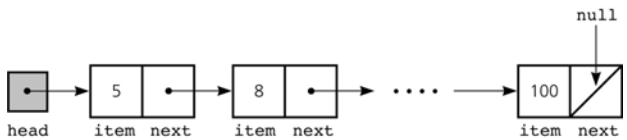


Figure 5-8

A ***head*** reference to a linked list

Reference-Based Linked Lists

- `head` reference variable can be assigned `null` without first using `new`
  - Following sequence results in a lost node

```
head = new Node(); // Don't really need to use new here
head = null; // since we lose the new Node object here
```

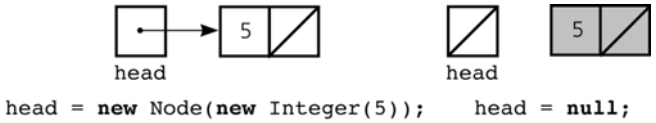
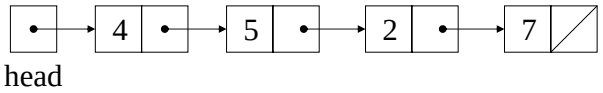


Figure 5-9

A lost node

Write the code

- To produce this linked list.



Programming with Linked Lists: Displaying the Contents of a Linked List

- `curr` reference variable
  - References the current node
  - Initially references the first node
- To display the data portion of the current node

```
System.out.println(curr.getItem());
```

- To advance the current position to the next node

```
curr = curr.getNext();
```


Displaying the Contents of a Linked List

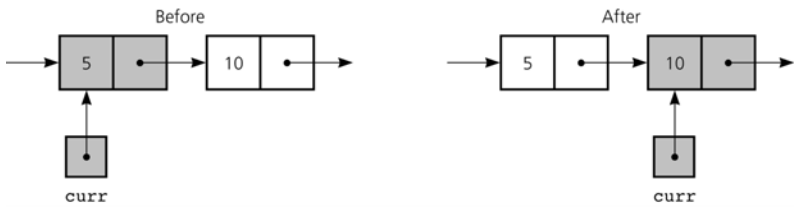


Figure 5-10

The effect of the assignment ***`curr = curr.getNext( )`***

Displaying the Contents of a Linked List

- To display all the data items in a linked list

```
for (Node curr = head; curr != null; curr =
    curr.getNext()) {
 System.out.println(curr.getItem());
}
```

This is an example of a list traversal. The code visits every node in the list in order.

Deleting a Specified Node from a Linked List

- To delete node N which `curr` references
  - Set `next` in the node that precedes N to reference the node that follows N

```
prev.setNext(curr.getNext());
```

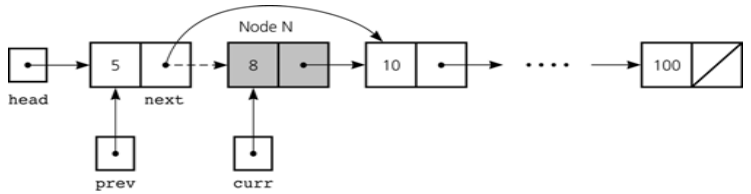


Figure 5-11

Deleting a node from a linked list

Deleting a Specified Node from a Linked List

- Deleting the first node is a special case

```
head = head.getNext();
```

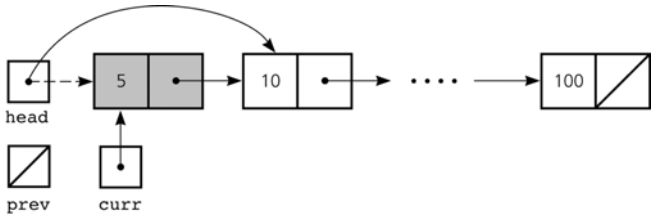


Figure 5-12

Deleting the first node

Deleting a Specified Node from a Linked List

- To return a node that is no longer needed to the system

```
curr.setNext(null);
curr = null;
```

- Three steps to delete a node from a linked list
  - Locate the node that you want to delete
  - Disconnect this node from the linked list by changing references
  - Return the node to the system

Inserting a Node into a Specified Position of a Linked List

- To create a node for the new item
  ```
  newNode = new Node(item);
  ```
- To insert a node between two nodes
  ```
  newNode.setNext(curr);
  prev.setNext(newNode);
  ```

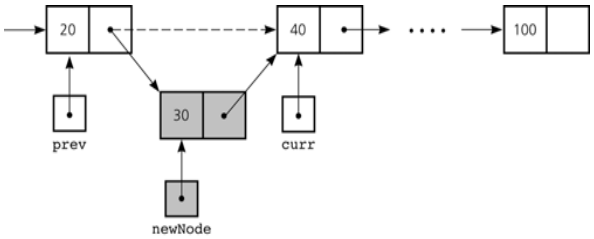


Figure 5-13

Inserting a new node into a linked list

Inserting a Node into a Specified Position of a Linked List

- To insert a node at the beginning of a linked list
  ```
  newNode.setNext(head);
  head = newNode;
  ```

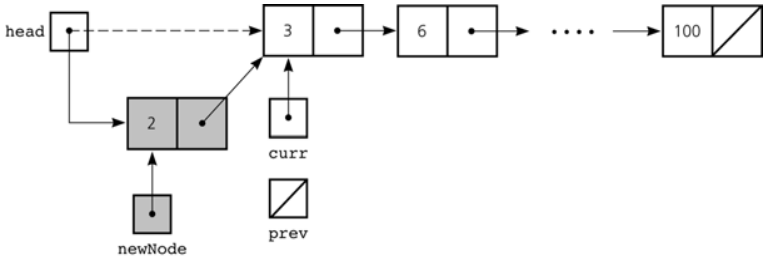


Figure 5-14

Inserting at the beginning of a linked list

Inserting a Node into a Specified Position of a Linked List

- Inserting at the end of a linked list is not a special case if `curr` is `null`
  ```
  newNode.setNext(curr);
  prev.setNext(newNode);
  ```

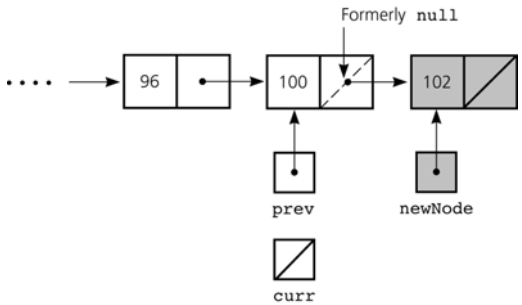


Figure 5-15

Inserting at the end of a linked list

Inserting a Node into a Specified Position of a Linked List

- Three steps to insert a new node into a linked list
  - Determine the point of insertion
  - Create a new node and store the new data in it
  - Connect the new node to the linked list by changing references

Determining `curr` and `prev`

- Determining the point of insertion or deletion for a **sorted** linked list of objects

```
for ( prev = null, curr = head;
      (curr != null) &&
         (newValue.compareTo(curr.getItem()) > 0);
       prev = curr, curr = curr.getNext() ) {
}
```