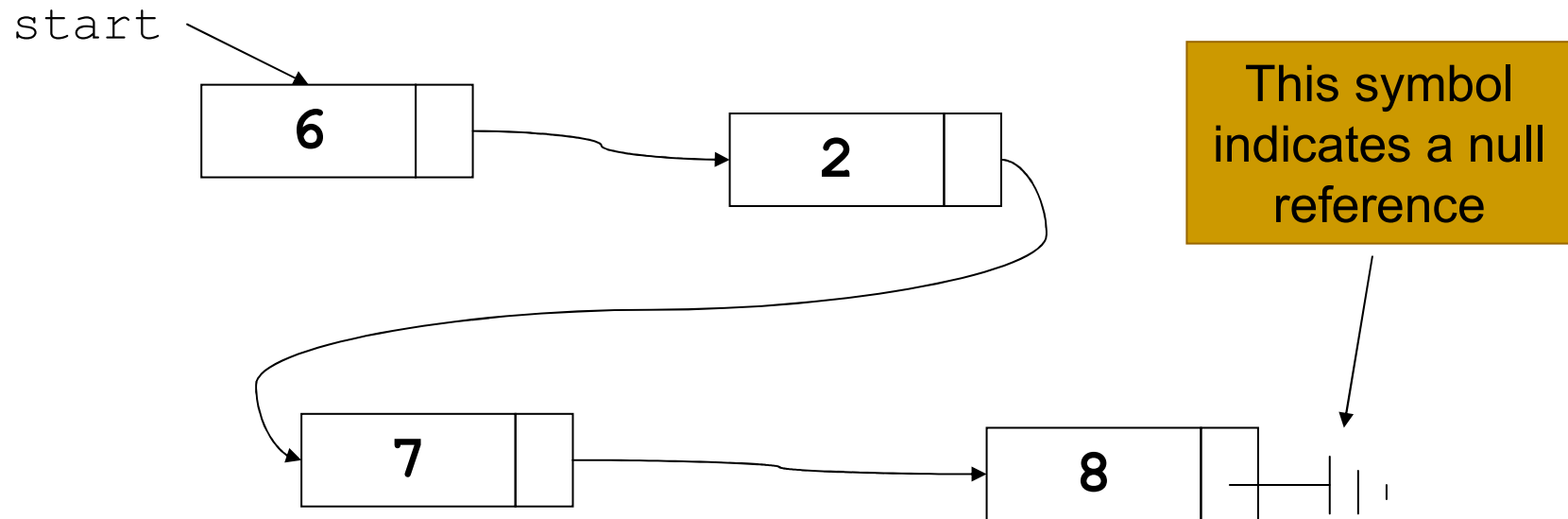


Linked Lists part II

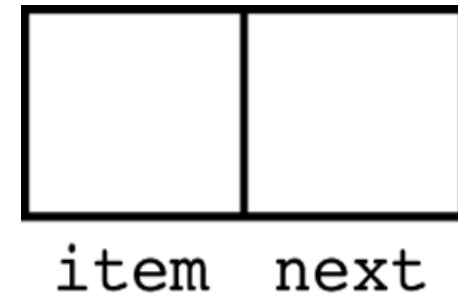
Linked Lists

- A linked list is a dynamic data structure consisting of nodes and links:



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, accessors,  
    // and mutators ...  
} // end class Node
```

Figure 5-5

A node

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

    public Node(Object newItem) {
        item = newItem;
        next = null;
    } // end constructor

    public Node(Object newItem, Node nextNode) {
        item = newItem;
        next = nextNode;
    } // end constructor

    public void setItem(Object newItem) {
        item = newItem;
    } // end setItem

    public Object getItem() {
        return item;
    } // end getItem

    public void setNext(Node nextNode) {
        next = nextNode;
    } // end setNext

    public Node getNext() {
        return next;
    } // end getNext
} // end class Node
```

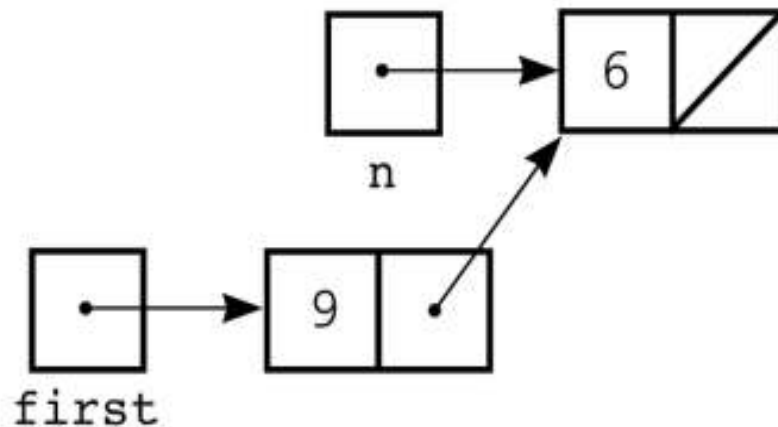
Reference-Based Linked Lists

- Using the Node class

```
Node n = new Node (new Integer(6));
```

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

```
Node n = new Node(new Integer(6));
```



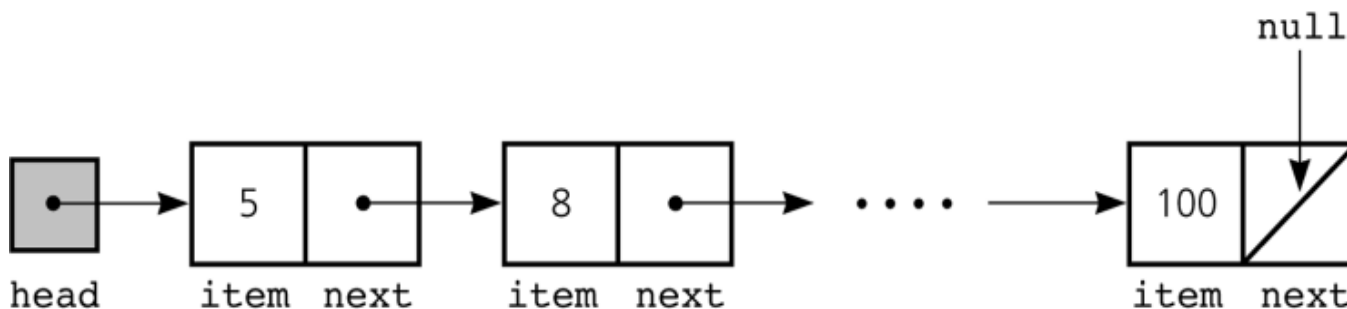
```
Node first = new Node(new Integer(9), n);
```

Figure 5-7

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

Reference-Based Linked Lists

- To manipulate a linked list we need to have references to the first and the last node in the list.
- Data field `next` in the last node is set to `null` therefore we can know when we reached the end of the list.
- `head` reference variable always points to the first node of the list.
 - References the list's first node
 - Always exists even when the list is empty



Traversing the 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

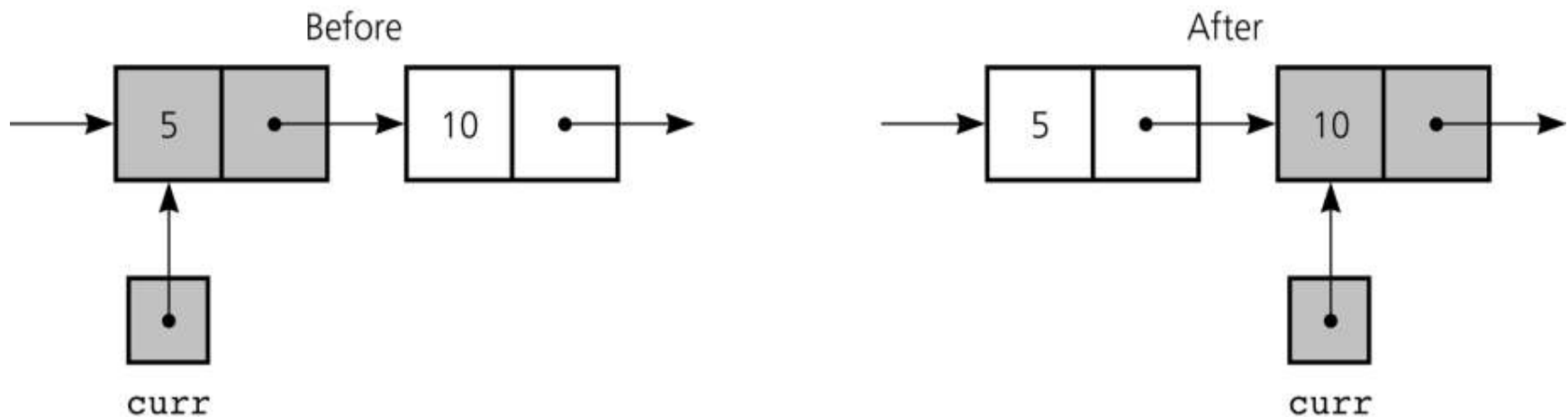


Figure 5-10

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());  
} // end for
```

- To compute the number of elements in a linked list.

```
for (int size=0, Node curr = head; curr != null;  
    curr = curr.getNext(), size++);
```

- A better option is to declare a size

Deleting a Specified Node from a Linked List

- To delete node N which `curr` references we also need a reference to the previous node.
 - Set `next` in the node that precedes N to reference the node that follows N

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

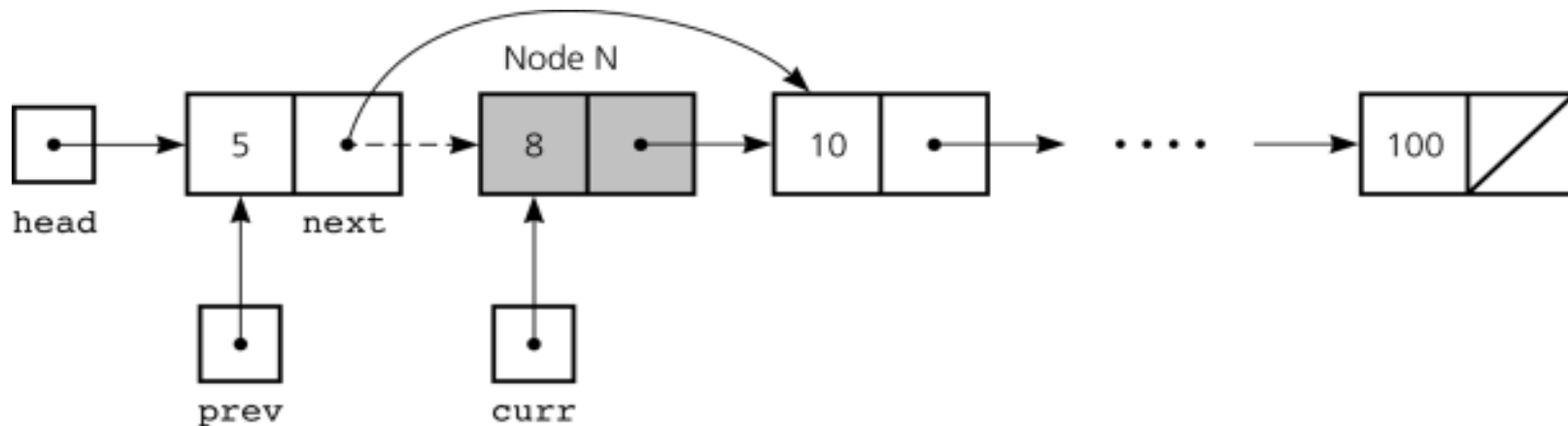


Figure 5-11

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();
```

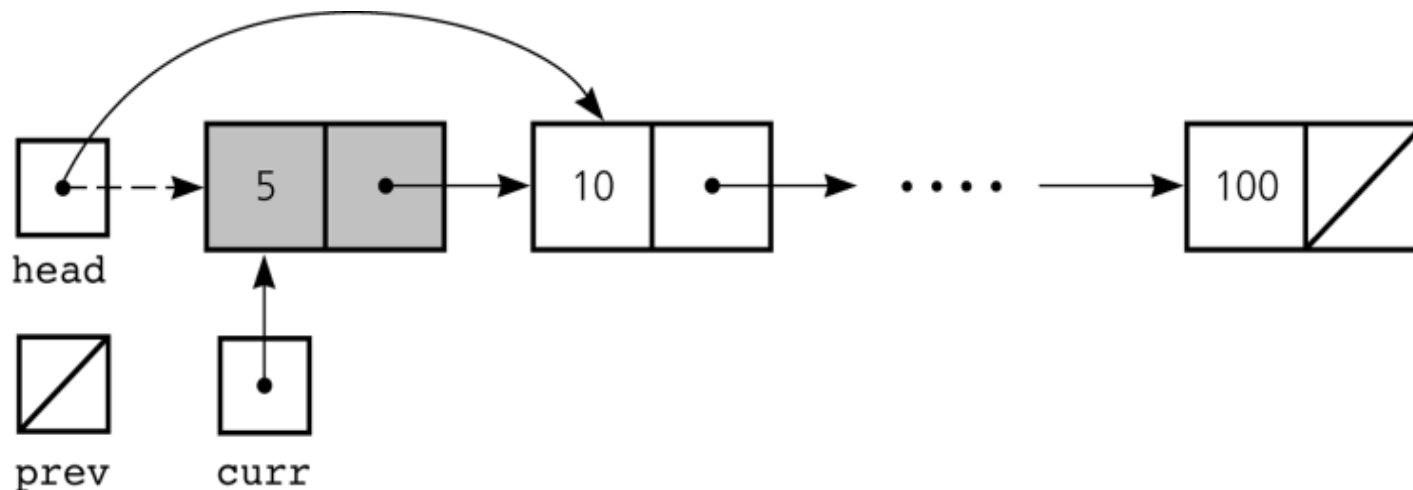


Figure 5-12

Deleting the first node

-
- How did the `cur` and `prev` come to the appropriate location?
 - It depends on the context for instance whether you are inserting by value (in a sorted list) or by position.
 - No matter what, the `cur` and `prev` references are not passed to the `delete` method.
 - Rather the method establishes those by traversing the list.
-

Delete pseudo code

```
//deleting a node that has value x.
void delete (valueType x){
    //first setting the cur and prev references appropriately.
    Node prev=null;
    Node cur=head;
    while (cur != null and cur.getItem() !=x){
        prev=cur;
        cur=cur.getNext();
    }
    if (cur != null){
        //delete the node at cur
        if(prev!=null)
            prev.setNext(cur.getNext());
        else
            head=head.getNext(); //the first node must be deleted.

        cur.setNext(null);
        cur=null;
    }
}
```

Deleting a Specified Node from a Linked List

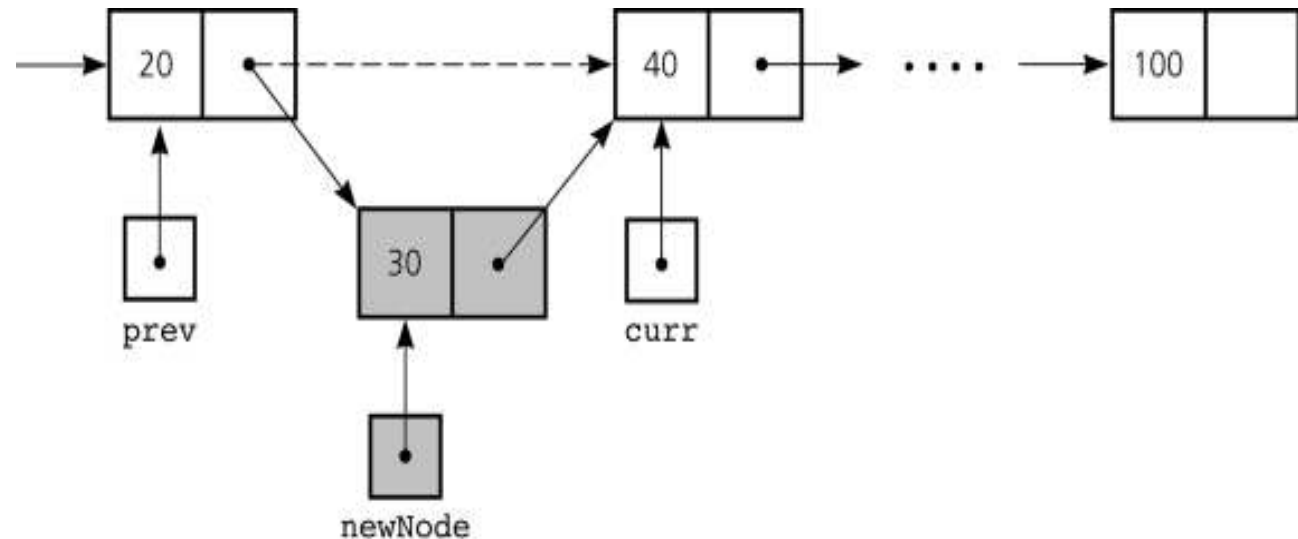
- To return a node that is no longer needed to the system

```
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);`



-
- How did the `cur` and `prev` come to the appropriate location?
 - It depends on the context for instance whether you are inserting by value (in a sorted list) or by position.
 - No matter what, the `cur` and `prev` references are not passed to the `delete` method.
 - Rather the method establishes those by traversing the 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;
```

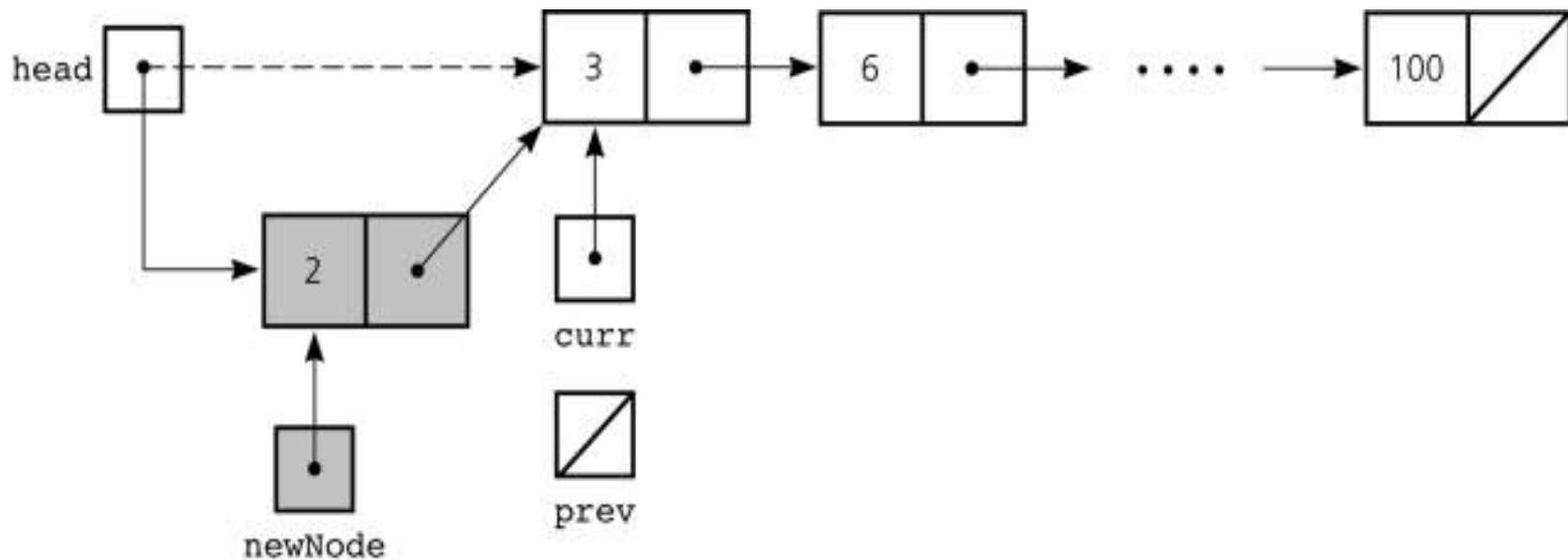


Figure 5-14

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);
```

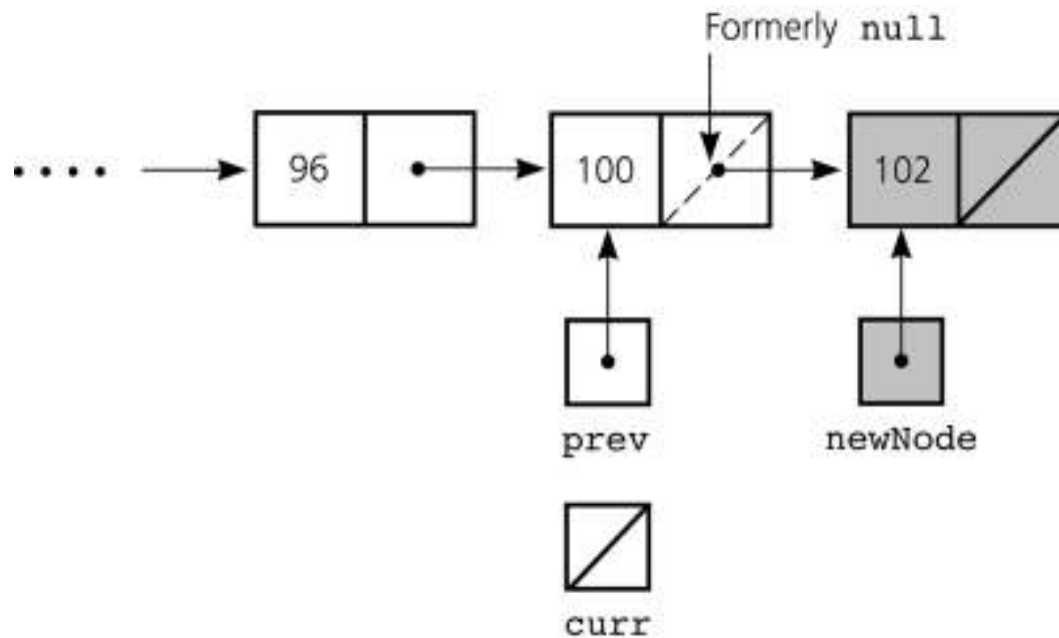


Figure 5-15

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
-

Insert pseudo code

```
//inserting a node that has value x in a sorted list.
void insert (valueType x){
    //first setting the cur and prev references appropriately.
    Node prev=null;
    Node cur=head;
    while (cur != null and cur.getItem() < x){
        prev=cur;
        cur=cur.getNext();
    }
    if (prev == null){
        //inset the node at the beginning of the list.
        Node newNode=new Node(x);
        newNode.setNext(head);
        head = newNode;
    }
    else{
        //insert the node between prev and cur
        Node newNode=new Node(x);
        newNode.setNext(cur);
        prev.setNext(newNode);
    }
}
```

A Reference-Based Implementation of the ADT List

- Default constructor
 - Initializes the data fields `numItems` and `head`
 - List operations
 - Public methods
 - `isEmpty`
 - `size`
 - `add`
 - `remove`
 - `get`
 - `removeAll`
 - Private method
 - `find`
-

Reference-based Implementation of ADT List

```
public boolean isEmpty() {  
    return numItems == 0;  
}    // end isEmpty
```

```
public int size() {  
    return numItems;  
}    // end size
```

Reference-based Implementation of ADT List

```
private Node find(int index) {  
    // -----  
    // Locates a specified node in a linked list.  
    // Precondition: index is the number of the desired  
    // node. Assumes that 1 <= index <= numItems+1  
    // Postcondition: Returns a reference to the desired  
    // node.  
    // -----  
    Node curr = head;  
    for (int skip = 1; skip < index; skip++) {  
        curr = curr.getNext();  
    } // end for  
    return curr;  
} // end find
```

Reference-based Implementation of ADT List

```
public Object get(int index)
    throws ListIndexOutOfBoundsException {
    if (index >= 1 && index <= numItems) {
        // get reference to node, then data in node
        Node curr = find(index);
        Object dataItem = curr.getItem();
        return dataItem;
    }
    else {
        throw new ListIndexOutOfBoundsException(
            "List index out of bounds on get");
    } // end if
} // end get
```

Reference-based Implementation of ADT List

```
public void add(int index, Object item)
    throws ListIndexOutOfBoundsException {
    if (index >= 1 && index <= numItems+1) {
        if (index == 1) {
            // insert the new node containing item at
            // beginning of list
            Node newNode = new Node(item, head);
            head = newNode;
        }
        else {
            Node prev = find(index-1);
            // insert the new node containing item after
            // the node that prev references
            Node newNode = new Node(item, prev.getNext());
            prev.setNext(newNode);
        } // end if
        numItems++;
    }
    else {
        throw new ListIndexOutOfBoundsException(
            "List index out of bounds on add");
    } // end if
} // end add
```

Reference-based Implementation of ADT List

```
public void remove(int index)
    throws ListIndexOutOfBoundsException {
    if (index >= 1 && index <= numItems) {
        if (index == 1) {
            // delete the first node from the list
            head = head.getNext();
        }
        else {
            Node prev = find(index-1);
            // delete the node after the node that prev
            // references, save reference to node
            Node curr = prev.getNext();
            prev.setNext(curr.getNext());
        } // end if
        numItems--;
    } // end if
    else {
        throw new ListIndexOutOfBoundsException(
            "List index out of bounds on remove");
    } // end if
} // end remove
```

Comparing search efficiency in a sorted list implemented by arrays vs linked list.

- Assume you want to implement the search in the sorted using binary search method.

```
int binarySearch(valueType x){
    int left=0, right=size();
    while (left <= right){
        int middle=(left+right)/2;
        Node m=get(middle);
        if (m.getValue()==x)
            return middle;
        if (m.getValue() > x)
            right = middle-1;
        else
            right = middle+1;
    }
    return -1; //the value is not found.
}
```

Binary search is impractical in a linked list.

- The maximum number of times the while loop iterates is $O(\log n)$ where n is the size of the list.
 - In an array based implementation of the list the `get(middle)` operation takes a constant time therefore the binarySearch is of $O(\log n)$
 - In a linked list implementation the `get(middle)` operation takes $O(n)$ therefore the binary search is of $O(n \cdot \log n)$
-

Variations of the Linked List:

Tail References

■ tail references

- Remembers where the end of the linked list is
- To add a node to the end of a linked list

```
tail.setNext(new Node(request, null));
```

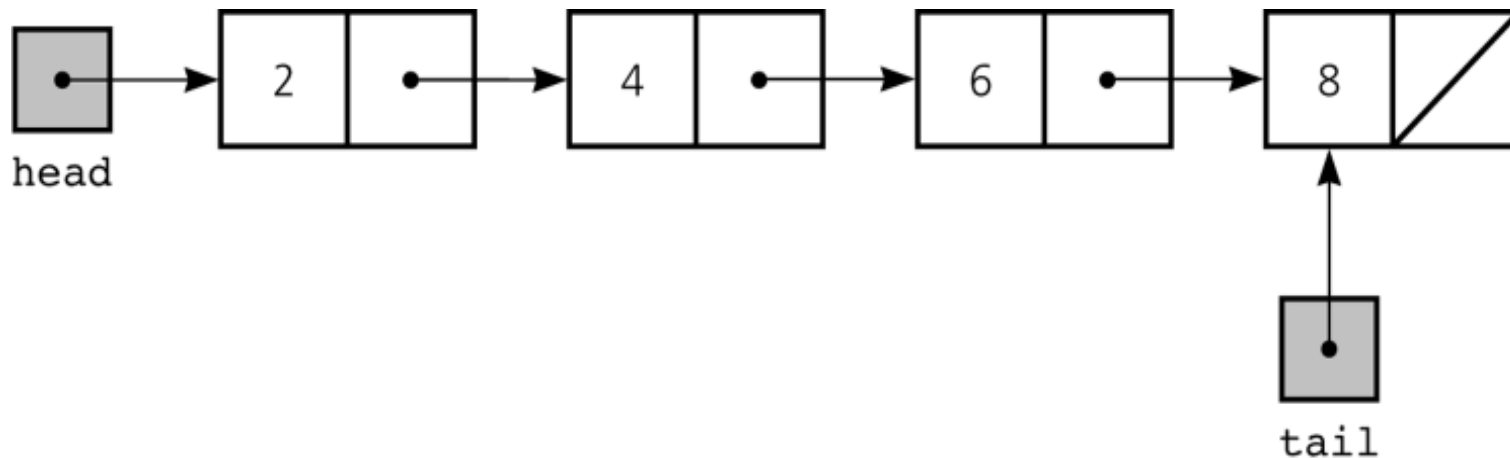


Figure 5-22

A linked list with *head* and *tail* references

Circular Linked List

- Last node references the first node
- Every node has a successor

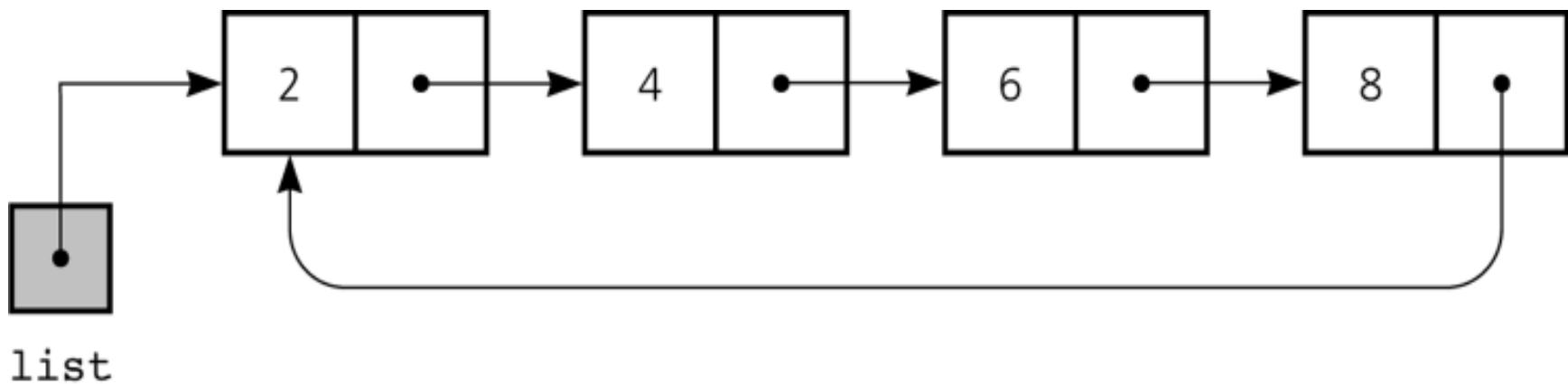


Figure 5-23

A circular linked list

Circular Linked List

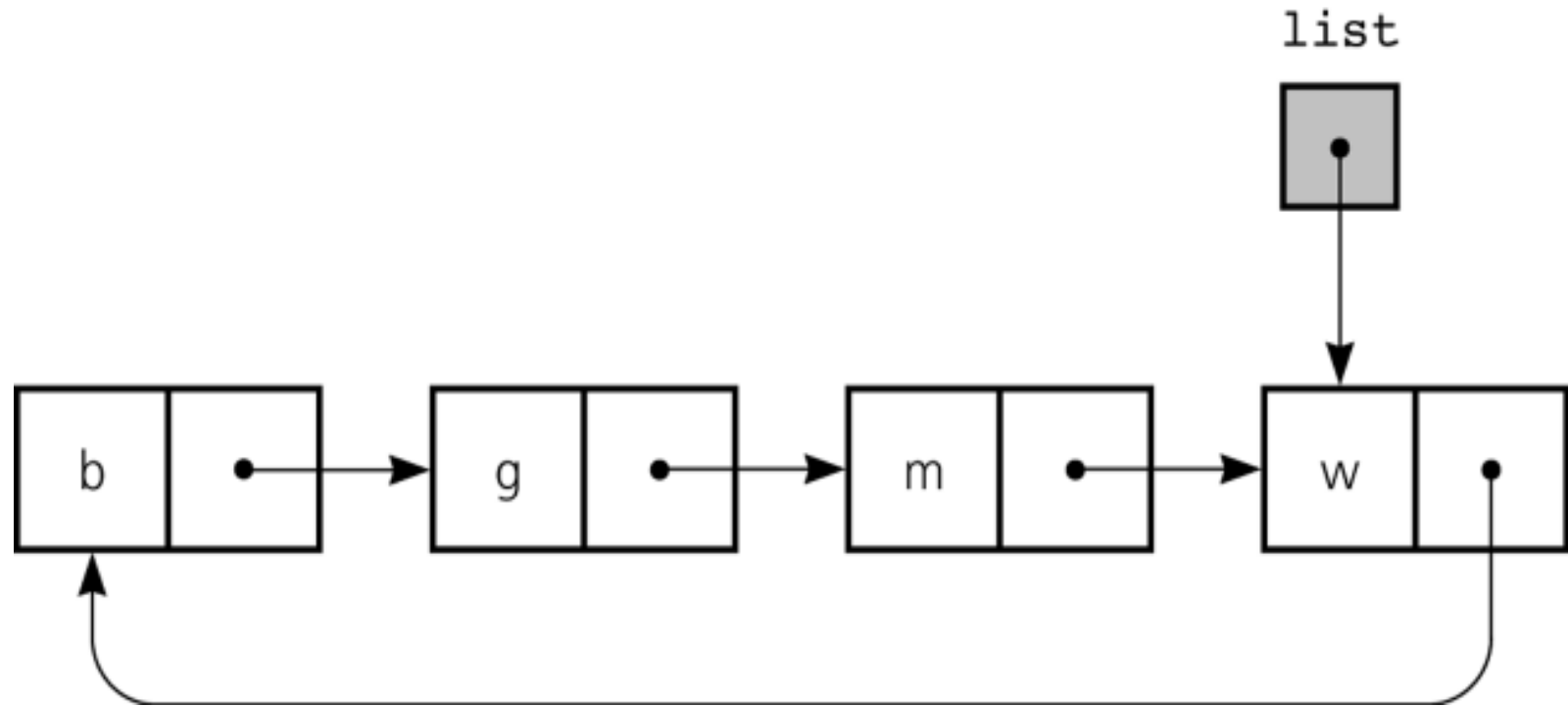


Figure 5-24

A circular linked list with an external reference to the last node

Dummy Head Nodes

- Dummy head node
 - Always present, even when the linked list is empty
 - Insertion and deletion algorithms initialize `prev` to reference the dummy head node, rather than `null`

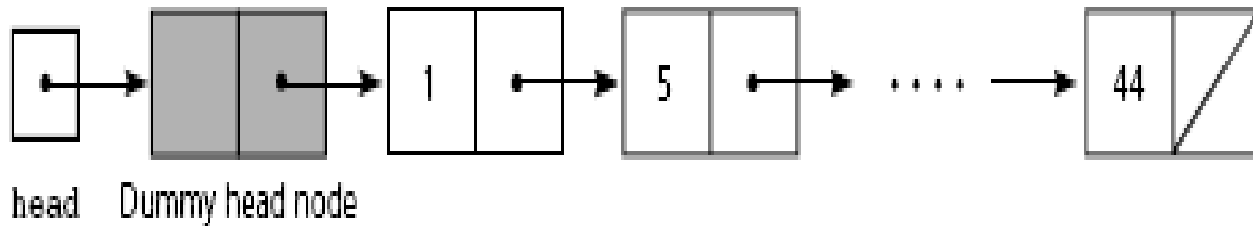


Figure 5-25

A dummy head node

Doubly Linked List

- Each node references both its predecessor and its successor
- Dummy head nodes are useful in doubly linked lists

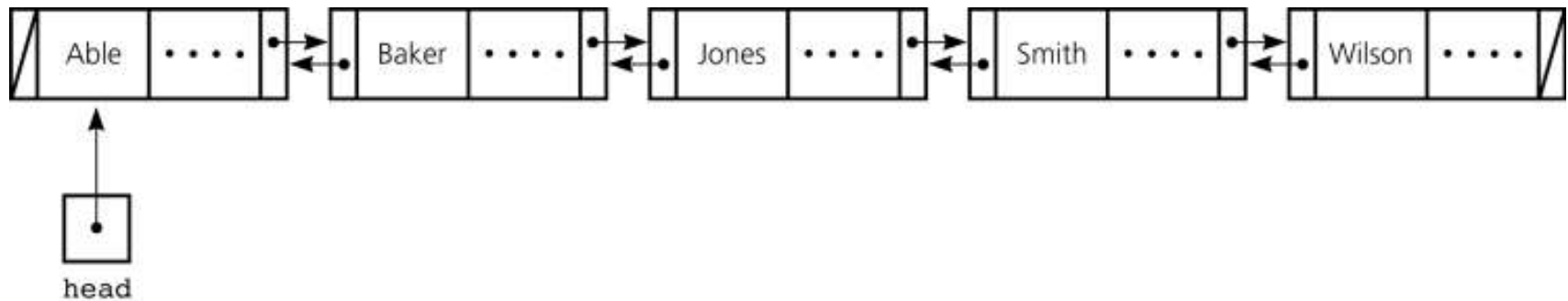


Figure 5-26

A doubly linked list

Doubly Linked List

- To delete the node that `curr` references

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

```
curr.getNext().setPrecede(curr.getPrecede());
```

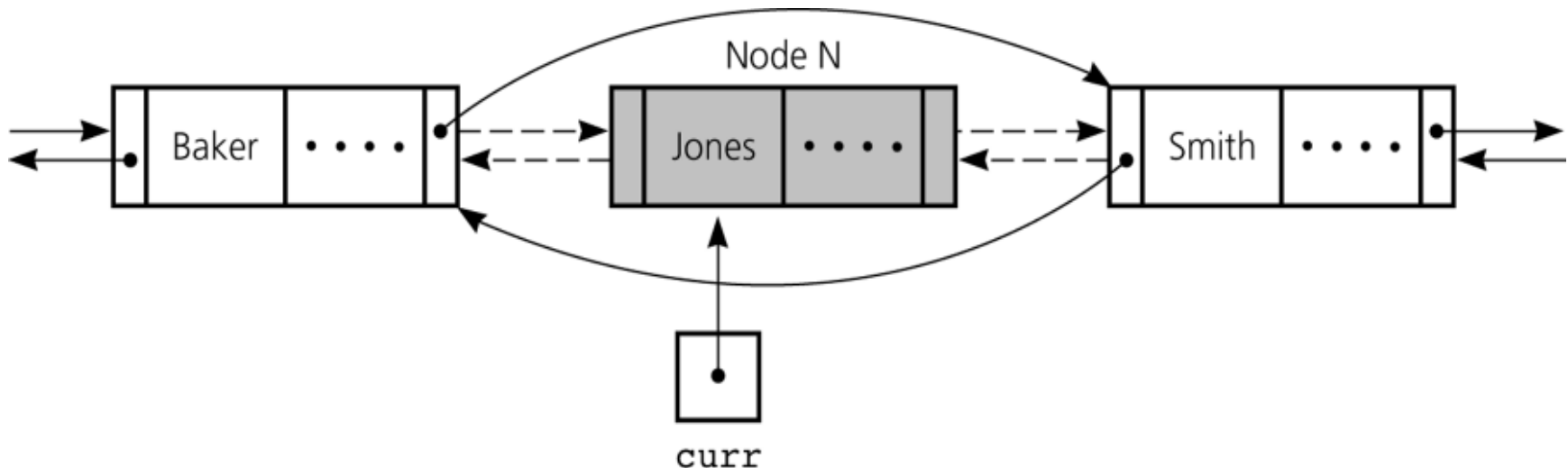


Figure 5-28

Reference changes for deletion

Doubly Linked List

- To insert a new node that `newNode` references before the node referenced by `curr`

```
newNode.setNext(curr);  
newNode.setPrecede(curr.getPrecede());  
curr.setPrecede(newNode);  
newNode.getPrecede().setNext(newNode);
```

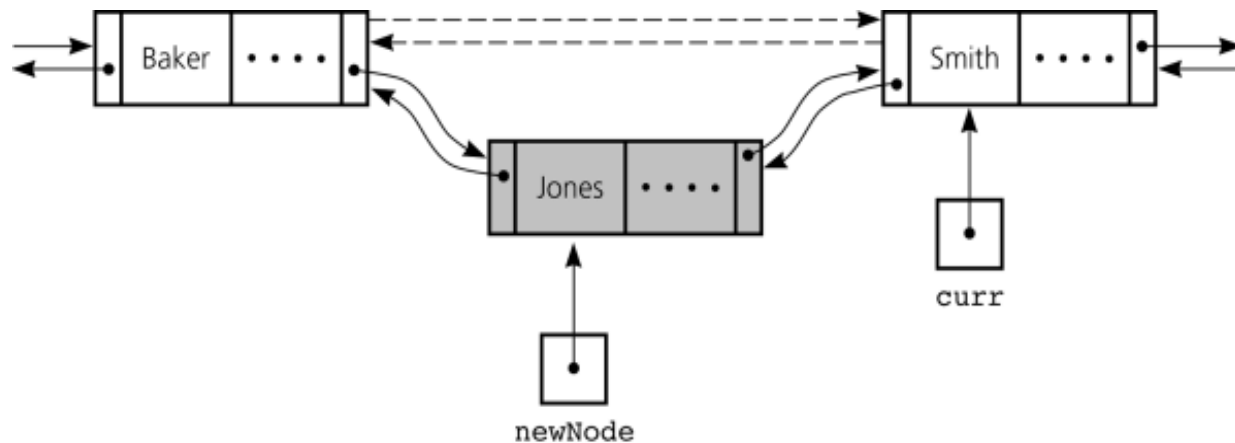


Figure 5-29

Reference changes
for insertion

Eliminating special cases in Doubly Linked List

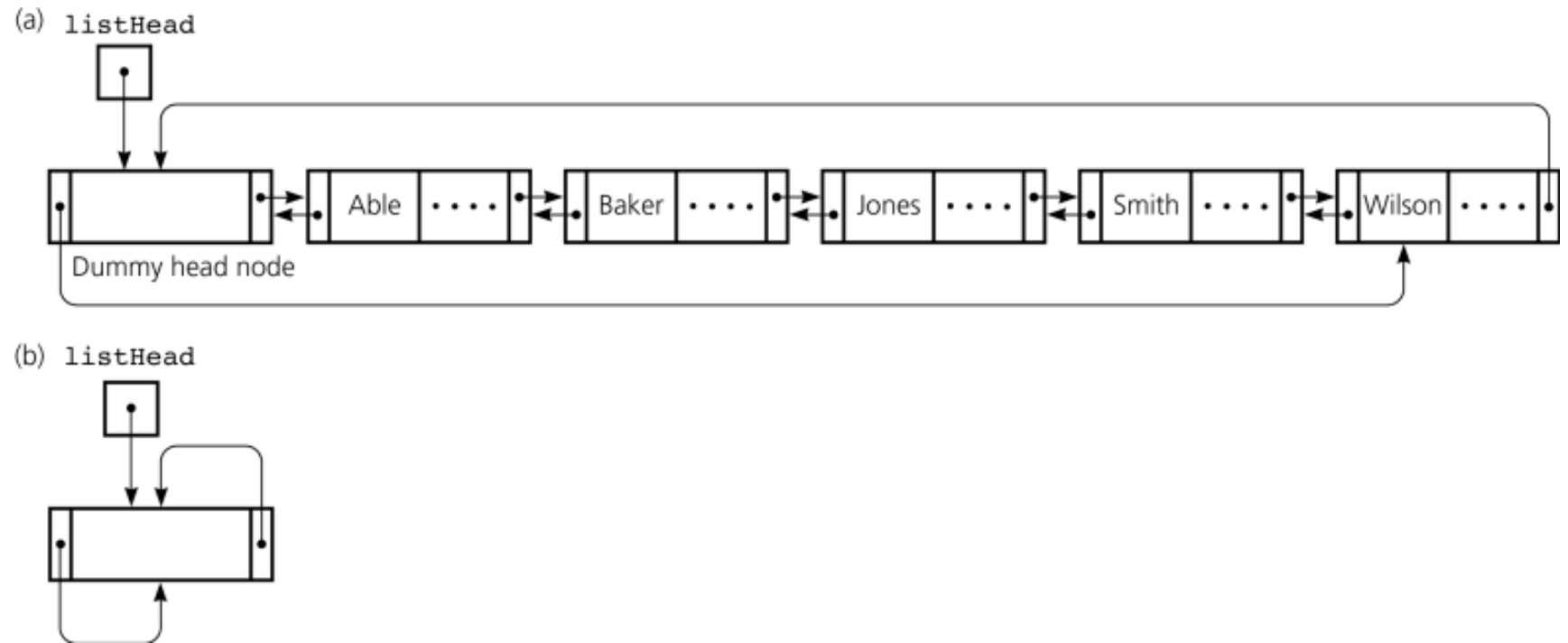


Figure 5-27

a) A circular doubly linked list with a dummy head node; b) an empty list with a dummy head node

Doubly Linked List

- Circular doubly linked list
 - `precede` reference of the dummy head node references the last node
 - `next` reference of the last node references the dummy head node
 - Eliminates special cases for insertions and deletions
-

Doubly Linked List - benefits

- Fast insertions and deletions at both ends of the list
- To perform deletion and insertion, we need reference to the current node (to node after place where we are inserting).

(Usual implementation of LinkedLists)
